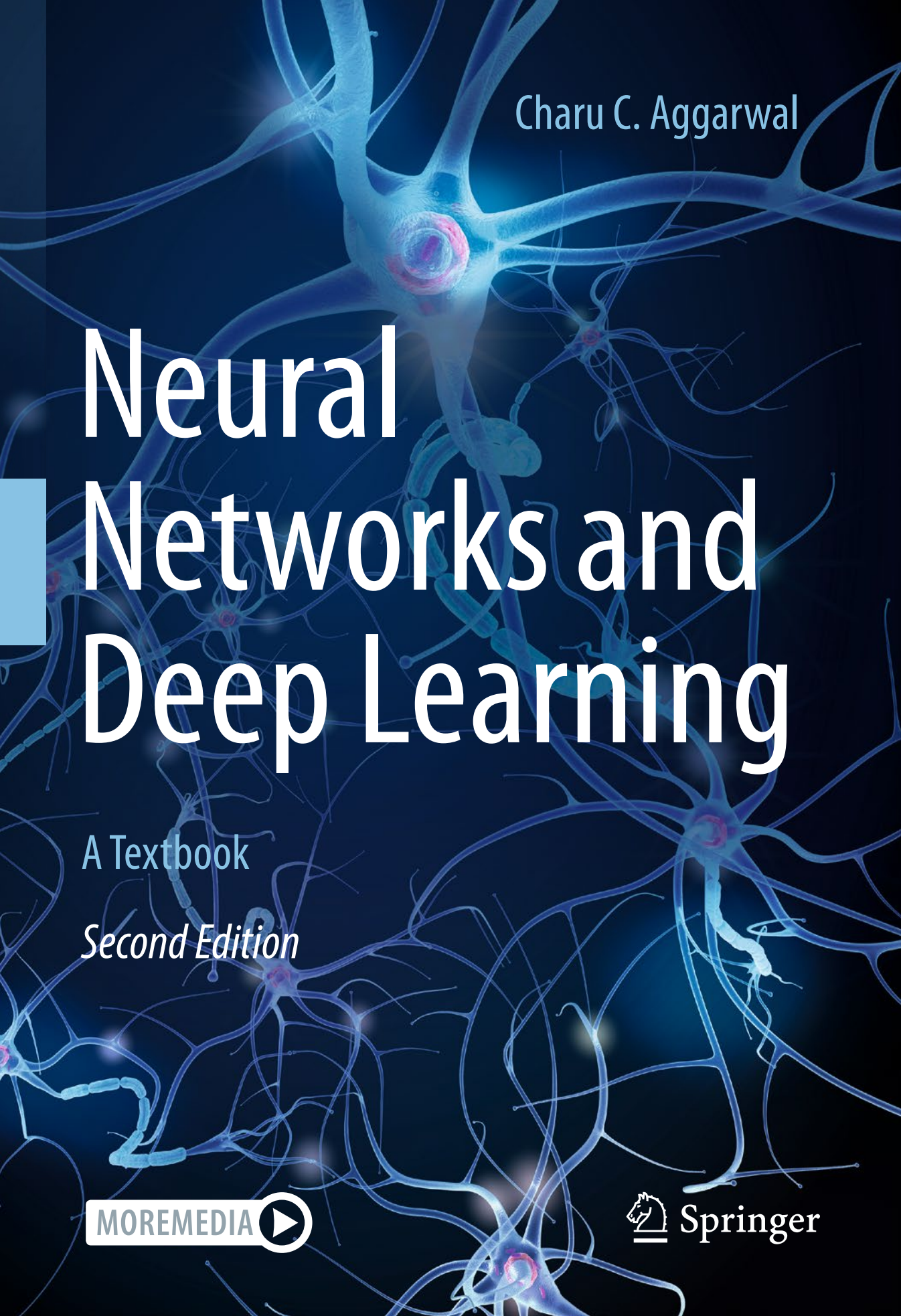

Charu C. Aggarwal
Neural Networks and Deep Learning
A Textbook
Second Edition
MOREMEDIA
Springer

Neural Networks and Deep Learning

Charu C. Aggarwal

Neural Networks and Deep Learning

A Textbook

Second Edition

Springer

Charu C. Aggarwal
IBM T. J. Watson Research Center
International Business Machines
Yorktown Heights, NY, USA

ISBN 978-3-031-29641-3 ISBN 978-3-031-29642-0 (eBook)
https://doi.org/10.1007/978-3-031-29642-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my wife Lata, my daughter Sayani,
and my late parents Dr. Prem Sarup and Mrs. Pushplata Aggarwal.

Preface

"Any A.I. smart enough to pass a Turing test is smart enough to know to fail it."–***Ian McDonald

Neural networks were developed to simulate the human nervous system for machine learning tasks by treating the computational units in a learning model in a manner similar to human neurons. The grand vision of neural networks is to create artificial intelligence by building machines whose architecture simulates the computations in the human nervous system. Although the biological model of neural networks is an exciting one and evokes comparisons with science fiction, neural networks have a much simpler and mundane mathematical basis than a complex biological system. The neural network abstraction can be viewed as a modular approach of enabling learning algorithms that are based on continuous optimization on a computational graph of mathematical dependencies between the input and output. These ideas are strikingly similar to classical optimization methods in control theory, which historically preceded the development of neural network algorithms.

Neural networks were developed soon after the advent of computers in the fifties and sixties. Rosenblatt's perceptron algorithm was seen as a fundamental cornerstone of neural networks, which caused an initial period of euphoria — it was soon followed by disappointment as the initial successes were somewhat limited. Eventually, at the turn of the century, greater data availability and increasing computational power lead to increased successes of neural networks, and this area was reborn under the new label of "deep learning." Although we are still far from the day that artificial intelligence (AI) is close to human performance, there are specific domains like image recognition, self-driving cars, and game playing, where AI has matched or exceeded human performance. It is also hard to predict what AI might be able to do in the future. For example, few computer vision experts would have thought two decades ago that any automated system could ever perform an intuitive task like categorizing an image more accurately than a human. The large amounts of data available in recent years together with increased computational power have enabled experimentation with more sophisticated and deep neural architectures than was previously possible. The resulting success has changed the broader perception of the potential of deep learning. This book discusses neural networks from this modern perspective. The chapters of the book are organized as follows:

1. *The basics of neural networks:* Chapters 1, 2, and 3 discuss the basics of neural network design and the backpropagation algorithm. Many traditional machine learning models

can be understood as special cases of neural learning. Understanding the relationship between traditional machine learning and neural networks is the first step to understanding the latter. The simulation of various machine learning models with neural networks is provided in Chapter 3. This will give the analyst a feel of how neural networks push the envelope of traditional machine learning algorithms.

2. *Fundamentals of neural networks:* Although Chapters 1, 2, and 3 provide an overview of the training methods for neural networks, a more detailed understanding of the training challenges is provided in Chapters 4 and 5. Chapters 6 and 7 present radial-basis function (RBF) networks and restricted Boltzmann machines.

3. *Advanced topics in neural networks:* A lot of the recent success of deep learning is a result of the specialized architectures for various domains, such as recurrent neural networks and convolutional neural networks. Chapters 8 and 9 discuss recurrent and convolutional neural networks. Graph neural networks are discussed in Chapter 10. Several advanced topics like deep reinforcement learning, attention mechanisms, neural Turing mechanisms, and generative adversarial networks are discussed in Chapters 11 and 12.

We have included some "forgotten" architectures like RBF networks and Kohonen self-organizing maps because of their potential in many applications. An application-centric view is highlighted throughout the book in order to give the reader a feel for the technology.

What Is New in The Second Edition

The second edition has focused on improving the presentations in the first edition and also on adding new material. Significant changes have been made to almost all the chapters to improve presentation and add new material where needed. In some cases, the material in different chapters has been reorganized and reordered in order to improve exposition. The discussion of training challenges with depth has been separated from the backpropagation chapter, so that both topics could be discussed in greater detail. Second-order methods have been explained with greater clarity and examples. Chapter 10 on graph neural networks is entirely new. The discussion on GRUs in Chapter 8 has been greatly enhanced. New convolutional architectures using attention, such as Squeeze-and-Excitation Networks, are discussed in Chapter 9. The chapter on reinforcement learning has been significantly reorganized in order to provide a clearer exposition. The Monte Carlo sampling approach for reinforcement learning is discussed in greater detail in its own dedicated section. Chapter 12 contains expanded discussions on attention mechanisms for graphs and computer vision, transformers, pre-trained language models, and adversarial learning.

Notations

Throughout this book, a vector or a multidimensional data point is annotated with a bar, such as $\overline{X}$ or $\overline{y}$. A vector or multidimensional point may be denoted by either small letters or capital letters, as long as it has a bar. Vector dot products are denoted by centered dots, such as $\overline{X} \cdot \overline{Y}$. A matrix is denoted in capital letters without a bar, such as R. Throughout the book, the $n \times d$ matrix corresponding to the training data set is denoted by D, with n documents and d dimensions. The individual data points in D are therefore d-dimensional

row vectors. On the other hand, vectors with one component for each data point are usually n-dimensional column vectors. An example is the n-dimensional column vector $\overline{y}$ of class variables. An observed value y_i is distinguished from a predicted value $\hat{y}_i$ by a circumflex at the top of the variable.

Yorktown Heights, NY, USA Charu C. Aggarwal

Acknowledgments

Acknowledgements for the First Edition

I would like to thank my family for their love and support during the busy time spent in writing this book. I would also like to thank my manager Nagui Halim for his support during the writing of this book.

Several figures in this book have been provided by the courtesy of various individuals and institutions. The Smithsonian Institution made the image of the Mark I perceptron (cf. Figure 1.5) available at no cost. Saket Sathe provided the outputs in Chapter 8 for the tiny Shakespeare data set, based on code available/described in [243, 604]. Andrew Zisserman provided Figures 9.13 and 9.17 in the section on convolutional visualizations. Another visualization of the feature maps in the convolution network (cf. Figure 9.16) was provided by Matthew Zeiler. NVIDIA provided Figure 11.10 on the convolutional neural network for self-driving cars in Chapter 11, and Sergey Levine provided the image on self-learning robots (cf. Figure 11.9) in the same chapter. Alec Radford provided Figure 12.11, which appears in Chapter 12. Alex Krizhevsky provided Figure 9.9(b) containing *AlexNet*.

This book has benefitted from significant feedback and several collaborations that I have had with numerous colleagues over the years. I would like to thank Quoc Le, Saket Sathe, Karthik Subbian, Jiliang Tang, and Suhang Wang for their feedback on various portions of this book. Shuai Zheng provided feedback on the section on regularized autoencoders in Chapter 5. I received feedback on the sections on autoencoders from Lei Cai and Hao Yuan. Feedback on the chapter on convolutional neural networks was provided by Hongyang Gao, Shuiwang Ji, and Zhengyang Wang. Shuiwang Ji, Lei Cai, Zhengyang Wang and Hao Yuan also reviewed the Chapters 4 and 8, and suggested several edits. They also suggested the ideas of using Figures 9.6 and 9.7 for elucidating the convolution/deconvolution operations.

For their collaborations, I would like to thank Tarek F. Abdelzaher, Jinghui Chen, Jing Gao, Quanquan Gu, Manish Gupta, Jiawei Han, Alexander Hinneburg, Thomas Huang, Nan Li, Huan Liu, Ruoming Jin, Daniel Keim, Arijit Khan, Latifur Khan, Mohammad M. Masud, Jian Pei, Magda Procopiuc, Guojun Qi, Chandan Reddy, Saket Sathe, Jaideep Srivastava, Karthik Subbian, Yizhou Sun, Jiliang Tang, Min-Hsuan Tsai, Haixun Wang, Jianyong Wang, Min Wang, Suhang Wang, Joel Wolf, Xifeng Yan, Mohammed Zaki, ChengXiang Zhai, and Peixiang Zhao. I would also like to thank my advisor James B. Orlin for his guidance during my early years as a researcher.

I would like to thank Lata Aggarwal for helping me with some of the figures created using PowerPoint graphics in this book. My daughter, Sayani, was helpful in incorporating special effects (e.g., image color, contrast, and blurring) in several JPEG images used at various places in this book.

Acknowledgements for the Second Edition

The chapter on graph neural networks benefited from collaborations with Jiliang Tang, Lingfei Wu, and Suhang Wang. Shuiwang Ji, Meng Liu, and Hongyang Gao provided detailed comments on the chapter on graph neural networks as well as other parts of the book. Zhengyang Wang and Shuiwang Ji provided feedback on the new section on transformer networks. Yao Ma and Jiliang Tang provided the permission to use Figure 10.3. Yao Ma also provided detailed comments on the chapter on graph neural networks. Sharmishtha Dutta, a PhD student at RPI, proofread Chapter 10, and also pointed out the sections that needed further clarity. My manager, Horst Samulowitz, provided support during the writing of the second edition of this book.

Contents

Author Biography

Charu C. Aggarwal is a Distinguished Research Staff Member (DRSM) at the IBM T. J. Watson Research Center in Yorktown Heights, New York. He completed his undergraduate degree in Computer Science from the Indian Institute of Technology at Kanpur in 1993 and his Ph.D. from the Massachusetts Institute of Technology in 1996.

He has worked extensively in the field of data mining. He has published more than 400 papers in refereed conferences and journals and authored over 80 patents. He is the author or editor of 20 books, including textbooks on data mining, recommender systems, and outlier analysis. Because of the commercial value of his patents, he has thrice been designated a Master Inventor at IBM. He is a recipient of an IBM Corporate Award (2003) for his work on bio-terrorist threat detection in data streams, a recipient of the IBM Outstanding Innovation Award (2008) for his scientific contributions to privacy technology, and a recipient of two IBM Outstanding Technical Achievement Awards (2009, 2015) for his work on data streams/high-dimensional data. He received the EDBT 2014 Test of Time Award for his work on condensation-based privacy-preserving data mining. He is a recipient of the IEEE ICDM Research Contributions Award (2015) and ACM SIGKDD Innovation Award, which are the two most prestigious awards for influential research contributions in the field of data mining. He is also a recipient of the W. Wallace McDowell Award, which is the highest award given solely by the IEEE Computer Society across the field of Computer Science.

He has served as the general co-chair of the IEEE Big Data Conference (2014) and as the program co-chair of the ACM CIKM Conference (2015), the IEEE ICDM Conference (2015), and the ACM KDD Conference (2016). He served as an associate editor of the IEEE Transactions on Knowledge and Data Engineering from 2004 to 2008. He is an associate editor of the IEEE Transactions on Big Data, an action editor of the Data Mining and Knowledge Discovery Journal, and an associate editor of the Knowledge and Information Systems Journal. He has served or currently serves as the editor-in-chief of the ACM Transactions on Knowledge Discovery from Data as well as the ACM SIGKDD Explorations. He is also an editor-in-chief of ACM Books. He serves on the advisory board of the Lecture Notes on Social Networks, a publication by Springer. He has served as the vice-president of

the SIAM Activity Group on Data Mining and is a member of the SIAM industry committee. He received the ACM SIGKDD Service Award for the aforem3entioned contributions to running conferences and journals— this honor is the most prestigious award for services to the field of data mining. He is a fellow of the SIAM, ACM, and the IEEE, for "contributions to knowledge discovery and data mining algorithms."

Chapter 1

An Introduction to Neural Networks

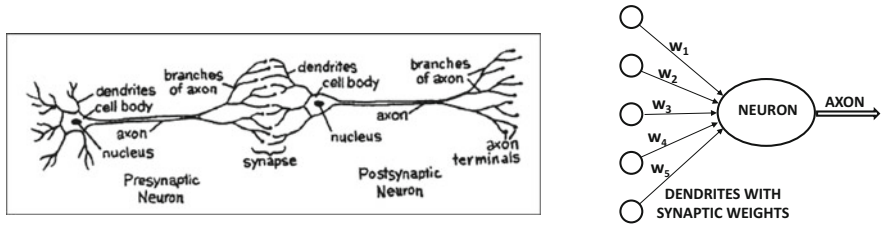
"Thou shalt not make a machine to counterfeit a human mind."–Frank Herbert

1.1 Introduction

Artificial neural networks are popular machine learning techniques that simulate the mechanism of learning in biological organisms. The human nervous system contains cells, which are referred to as *neurons*. The neurons are connected to one another with the use of *axons* and *dendrites*, and the connecting regions between axons and dendrites are referred to as *synapses*. These connections are illustrated in Figure 1.1(a). The strengths of synaptic connections often change in response to external stimuli. This change is how learning takes place in living organisms.

This biological mechanism is simulated in *artificial* neural networks, which contain computation units that are referred to as neurons. Throughout this book, we will use the term "neural networks" to refer to artificial neural networks rather than biological ones. The computational units are connected to one another through weights, which serve the same role as the strengths of synaptic connections in biological organisms. Each input to a neuron is scaled with a weight, which affects the function computed at that unit. This architecture is illustrated in Figure 1.1(b). An artificial neural network computes a function of the inputs by propagating the computed values from the input neurons to the output neuron(s) and using the weights as intermediate parameters. Learning occurs by changing the weights connecting the neurons. Just as external stimuli are needed for learning in biological organisms, the external stimulus in artificial neural networks is provided by the *training data* containing examples of input-output pairs of the function to be learned. For example, the training data might contain pixel representations of images (input) and their annotated labels (e.g., carrot, banana) as the output. These training data pairs are fed into the neural network by using the input representations to make predictions about the output labels. The training data provides feedback to the correctness of the weights in the neural network depending on how well the predicted output (e.g., probability of carrot) for a particular input matches

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_1



(a) Biological neural network (b) Artificial neural network

Figure 1.1: The synaptic connections between neurons. The image in (a) is from "*The Brain: Understanding Neurobiology Through the Study of Addiction* [621]." Copyright ©2000 by BSCS & Videodiscovery. All rights reserved. Used with permission.

the annotated output label in the training data. One can view the errors made by the neural network in the computation of a function as a kind of unpleasant feedback in a biological organism, leading to an adjustment in the synaptic strengths. Similarly, the weights between neurons are adjusted in a neural network in response to prediction errors. The goal of changing the weights is to modify the computed function to make the predictions more correct in future iterations. Therefore, the weights are changed carefully in a mathematically justified way so as to reduce the error in computation on that example. By successively adjusting the weights between neurons over many input-output pairs, the function computed by the neural network is refined over time so that it provides more accurate predictions. Therefore, if the neural network is trained with many different images of bananas, it will eventually be able to properly recognize a banana in an image it has not seen before. This ability to accurately compute functions of unseen inputs by training over a finite set of input-output pairs is referred to as *model generalization*. The primary usefulness of all machine learning models is gained from their ability to generalize their learning from seen training data to unseen examples.

The biological comparison is often criticized as a very poor caricature of the workings of the human brain; nevertheless, the principles of neuroscience have often been useful in designing neural network architectures. A different view is that neural networks are built as higher-level abstractions of the classical models that are commonly used in machine learning. In fact, the most basic units of computation in the neural network are inspired by traditional machine learning algorithms like *least-squares regression* and *logistic regression*. Neural networks gain their power by putting together many such basic units, and learning the weights of the different units jointly in order to minimize the prediction error. From this point of view, a neural network can be viewed as a *computational graph* of elementary units in which greater power is gained by connecting them in particular ways. When a neural network is used in its most basic form, without hooking together multiple units, the learning algorithms often reduce to classical machine learning models (see Chapter 3). The real power of a neural model over classical methods is unleashed when these elementary computational units are combined. By combining multiple units, one is increasing the power of the model to learn more complicated functions of the data than are inherent in the elementary models of basic machine learning. The way in which these units are combined also plays a role in the power of the architecture, and requires some understanding and insight from the analyst.

Humans versus Computers: Stretching the Limits of Artificial Intelligence

Humans and computers are inherently suited to different types of tasks. For example, computing the cube root of a large number is very easy for a computer, but it is extremely difficult for humans. On the other hand, a task such as recognizing the objects in an image is a simple matter for a human, but has traditionally been very difficult for an automated learning algorithm. It is only in recent years that deep learning has shown an accuracy on some of these tasks that exceeds that of a human. In fact, the recent results by deep learning algorithms that surpass human performance [194] in (some narrow tasks on) image recognition would not have been considered likely by most computer vision experts as recently as 20 years ago.

Many neural networks that have shown such extraordinary performance by connecting computational units in careful ways with a *deep architecture*. This performance mirrors the fact that biological neural networks gain much of their power from depth and careful connectivity as well. Unfortunately, biological networks are connected in ways we do not fully understand. In the few cases that the biological structure is understood at some level, significant breakthroughs have been achieved by designing artificial neural networks along those lines. A classical example of this type of architecture is the use of the *convolutional neural network* for image recognition. This architecture was inspired by Hubel and Wiesel's experiments [223] in 1959 on the organization of the neurons in the cat's visual cortex. The precursor to the convolutional neural network was the *neocognitron* [132], which was directly based on these results.

The human neuronal connection structure has evolved over millions of years to optimize survival-driven performance; survival is closely related to our ability to merge sensation and intuition in a way that is currently not possible with machines. Biological neuroscience [242] is a field that is still very much in its infancy, and only a limited amount is known about how the brain truly works. Therefore, it is fair to suggest that the biologically inspired success of convolutional neural networks might be replicated in other settings, as we learn more about how the human brain works [186].

A large part of the recent success of neural networks is explained by the fact that the increased data availability and computational power of modern computers has outgrown the limits of traditional machine learning algorithms, which fail to take full advantage of what is now possible. This situation is illustrated in Figure 1.2. The performance of traditional machine learning remains better at times for smaller data sets because of more

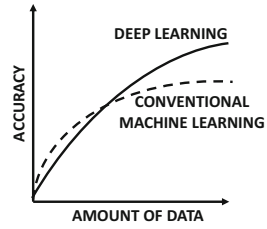


Figure 1.2: An illustrative comparison of the accuracy of a typical machine learning algorithm with that of a large neural network. Recent years have seen an increase in data availability and computational power, which has led to a "Cambrian explosion" in deep learning technology.

choices, greater ease of model interpretation, and the tendency to hand-craft interpretable features that incorporate domain-specific insights. With limited data, the best of a very wide diversity of models in machine learning will usually perform better than a single class of models (like neural networks). This is one reason why the potential of neural networks was not realized in the early years.

The "big data" era has been enabled by the advances in data collection technology; virtually everything we do today, including purchasing an item, using the phone, or clicking on a site, is collected and stored somewhere. Furthermore, the development of powerful Graphics Processor Units (GPUs) has enabled increasingly efficient processing on such large data sets. These advances largely explain the recent success of deep learning using algorithms that are only slightly adjusted from the versions that were available two decades back. Furthermore, these recent adjustments to the algorithms have been enabled by increased speed of computation, because reduced run-times enable efficient testing (and subsequent algorithmic adjustment). If it requires a month to test an algorithm, at most twelve variations can be tested in an year on a single hardware platform. This situation has historically constrained the intensive experimentation required for tweaking neural-network learning algorithms. The rapid advances associated with the three pillars of improved data, computation, and experimentation have resulted in an increasingly optimistic outlook about the future of deep learning. By the end of this century, it is expected that computers will have the power to train neural networks with as many neurons as the human brain. Although it is hard to predict what the true capabilities of artificial intelligence will be by then, our experience with computer vision should prepare us to expect the unexpected.

Chapter Organization

This chapter is organized as follows. The next section introduces single-layer and multilayer networks. The different components of neural networks are discussed in section 1.3. Multilayer neural networks are introduced in section 1.4. The importance of nonlinearity is discussed in section 1.5. Advanced topics in deep learning are discussed in section 1.6. Some notable benchmarks used by the deep learning community are discussed in section 1.7. A summary is provided in section 1.8.

The Basic Idea of Neural Networks

All neural networks compute functions from input nodes to output nodes. Each node has a variable inside it which is the result of a function computation or is fixed externally (in case of *input nodes*). A neural network is structured as a directed acyclic graph. The edges of the neural network are parameterized with weights, so that the functions computed at individual nodes are affected by the weights of the incoming edges as well as the variables in the nodes at the tails of these edges. The overall function computed by a network is result of cascading function computations at individual nodes; by setting the weights on the edges appropriately, one can compute almost any function from the input to the output. In a *single-layer network*, a set of inputs is directly connected to one or more output nodes, and the output nodes compute a function of their inputs and the weights on the edges. In multi-layer neural networks, the neurons are arranged in layered fashion, in which the input and output layers are separated by a group of hidden layers of nodes.

The goal of a neural network is to *learn* a function that relates one or more inputs to one or more outputs with the use of *training examples*. In other words, we only have examples of inputs and outputs, and the neural network "constructs" a function that relates the

inputs to the outputs. This "construction" is achieved by setting the weights on the edges in such a way that the computations from inputs to outputs in the neural network match the observed data. Setting the edge weights in a data-driven manner is at the heart of all neural network learning. This process is also referred to as *training*.

For simplicity, we first consider the case where there are d inputs and a single *binary* output drawn from $\{-1, +1\}$. Therefore, each training instance is of the form $(\overline{X}, y)$, where each $\overline{X} = [x_1, \ldots x_d]$ contains d feature variables, and $y \in \{-1, +1\}$ contains the *observed value* of the binary class variable. By "observed value" we refer to the fact that it is given to us as a part of the training data, and our goal is to predict the class variable for cases in which it is not observed. The variable y is referred to as the *target variable*, and it represents the all-important property that the machine learning algorithm is trying to predict. In other words, we want to learn the function $f(\cdot)$, where we have the following:

$$y = f_{\overline{W}}(\overline{X})$$

Here, $\overline{X}$ corresponds to the feature vector, and $\overline{W}$ is a weight vector that needs to be computed in a data-driven manner in order to "construct" the prediction function. The function $f(\overline{X})$ has been subscripted with a weight vector to show that it depends on $\overline{W}$ as well. Most machine learning problems reduce to the following optimization problem in one form or the other:

$$\text{Minimize}_{\overline{W}} \text{ Mismatching between } y \text{ and } f_{\overline{W}}(\overline{X})$$

The mismatching between predicted and observed values is penalized with the use of a *loss function*. For example, in a credit-card fraud detection application, the features in the vector $\overline{X}$ might represent various properties of a credit card transaction (e.g., amount and type of transaction), and the binary class variable $y \in \{1, -1\}$ might represent whether or not this transaction is fraudulent. Clearly, in this type of application, one would have historical cases in which the class variable is observed (i.e., transactions together with their fraudulent indicator), and other (current) cases of transactions in which the class variable has not yet been observed but needs to be predicted. This second type of data is referred to as the *testing data*. Therefore, one must first construct the function $f(\overline{X})$ with the use of the training data (in an indirect way by deriving the weights of the neural network), and then use this function in order to predict the value of y using this function for cases in which the target variable y is not known. In the next section, we will discuss a neural network with a single computational layer that can be trained with this type of data.

1.2 Single Computational Layer: The Perceptron

The simplest neural network is referred to as the perceptron. This neural network contains a single input layer and an output node. The basic architecture of the perceptron is shown in Figure 1.3(a). Since the training data has d inputs and a single output, we assume that the neural network has d inputs denoted by the row vector $\overline{X}$ (which is the argument of the function of to be learned) and a single output y. The input layer contains d nodes that transmit the d features $x_1 \ldots x_d$ contained in the row vector $\overline{X} = [x_1 \ldots x_d]$. We use row vectors for features (instead of the usual convention of using column vectors) since feature vectors are often extracted from rows of data matrices. The input layer is incident edges of weight $w_1 \ldots w_d$, contained in the column vector $\overline{W} = [w_1 \ldots w_d]^T$, and all these edges are incident on a single output node. The input layer does not perform any computation

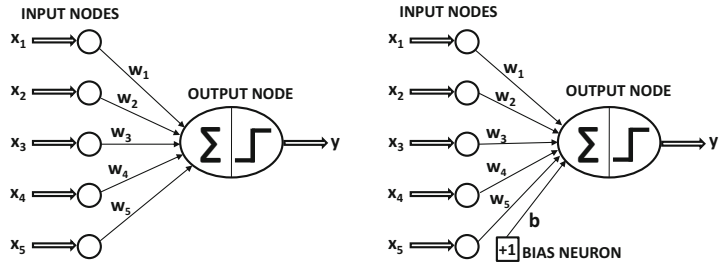
in its own right. The linear function $\overline{W} \cdot \overline{X}^T = \sum_{i=1}^{d} w_i x_i$ is computed at the output node. Subsequently, the sign of this real value is used in order to predict the dependent variable of $\overline{X}$. Therefore, the prediction $\hat{y}$ is computed as follows:

$$\hat{y} = f(\overline{X}) = \text{sign}\{\overline{W} \cdot \overline{X}^T\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\} \tag{1.1}$$

The sign function maps a real value to either $+1$ or -1, which is appropriate for binary classification. Note the circumflex on top of the variable $\hat{y}$ to indicate that it is a predicted value rather than the (original) observed value y. One can already see that the neural network creates a basic form of the function $f(\cdot)$ to be learned, although the weights $\overline{W}$ are unknown and need to be inferred in a data-driven manner from the training data. In most cases, a *loss function* is to used to quantify a nonzero (positive) cost when the observed value y of the class variable is different from the predicted value $\hat{y}$. The weights of the neural network are learned in order to minimize the aggregate loss over all training instances.

Initially, the weights in column vector $\overline{W}$ are unknown and they may be set randomly. As a result, the prediction $\hat{y}$ will be random for a given input $\overline{X}$, and it may not match the observed value y. The goal of the learning process is to use these errors to modify the weights, so that the neural network predictions become more accurate. Therefore, a neural network starts with a basic form of the function $f(\overline{X})$ while allowing some ambiguities in the form of *parametrization*. By learning the parameters, also referred to as the weights, one also learns the function computed by the neural network. This is a general principle in machine learning, where the basic structure of a function relating two or more variables is chosen, and some parameters are left unspecified. These parameters are then learned in a data-driven manner, so that the functional relationships between the two variables are consistent with the observed data. This process requires the formulation of an optimization problem.

The architecture of the perceptron is shown in Figure 1.3(a), in which a single input layer transmits the features to the output node. The edges from the input to the output contain the weights $w_1 \ldots w_d$ with which the features are multiplied and added at the output node. Subsequently, the sign function is applied in order to convert the aggregated value into a class label. The sign function serves the role of an *activation function*, and we will see many other examples of this type of function later in this chapter. The value of the variable in a node in a neural network is also sometimes referred to as an *activation*. It is noteworthy that the perceptron contains two layers, although the input layer does not



(a) Perceptron without bias (b) Perceptron with bias

Figure 1.3: The basic architecture of the perceptron

perform any computation and only transmits the feature values. This is true across all neural architectures, where the input layer only performs a transmission role. The input layer is not included in the count of the number of layers in a neural network. Since the perceptron contains a single *computational* layer, it is considered a single-layer network.

At the time that the perceptron algorithm was proposed by Rosenblatt [421], the weights were learned using a heuristic update process with actual hardware circuits, and it was not presented in terms of a formal notion of optimization of loss functions in machine learning (as is common today). However, the goal was always to minimize the error in prediction, even if a formal optimization formulation was not presented. The perceptron algorithm was, therefore, heuristically designed to minimize the number of misclassifications. The training algorithm works by feeding each input data instance $\overline{X}$ into the network one by one (or in small batches) to create the prediction $\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}^T\}$. When this prediction is different from the observed class $y \in \{-1, +1\}$, the weights are updated using the error value $(y - \hat{y})$ so that this error is less likely to occur after the update. Specifically, when the data point $\overline{X}$ is fed into the network, the weight vector $\overline{W}$ (which is a column vector) is updated as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha(y - \hat{y})\overline{X}^T \tag{1.2}$$

The (heuristic) rationale for the above update is that it always increases the dot product of the weight vector and $\overline{X}^T$ by a quantity proportional to $(y - \hat{y})$, which tends to improve the prediction for that training instance. The parameter α regulates the *learning rate* of the neural network, although it can be set to 1 in the special case of the perceptron (as it only scales the weights). The perceptron algorithm repeatedly cycles through all the training examples in random order and iteratively adjusts the weights until convergence is reached. A single training data point may be cycled through many times. Each such cycle is referred to as an *epoch*.

Since the value of $(y - \hat{y}) \in \{-2, 0, +2\}$ is always equal to $2y$ in cases where $y \neq \hat{y}$, one can write the perceptron update as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha y \overline{X}^T \tag{1.3}$$

Note that a factor of 2 has been absorbed in the learning rate, and the update is *only* performed for examples where $y \neq \hat{y}$. For correctly classified training instances, no update is performed.

The type of model proposed in the perceptron is a *linear model*, in which the equation $\overline{W} \cdot \overline{X}^T = 0$ defines a linear hyperplane. Here, $\overline{W} = [w_1 \ldots w_d]^T$ is a d-dimensional vector that is perpendicular to this hyperplane. Furthermore, the value of $\overline{W} \cdot \overline{X}^T$ is positive for values of values of $\overline{X}$ on one side of the hyperplane, and it is negative for values of $\overline{X}$ on the other side. This type of model performs particularly well when the data is *linearly separable*. Linear separability refers to the case in which the two classes can be separated with a linear hyperplane. Examples of linearly separable and inseparable data are shown in Figure 1.4. In linearly separable cases, a weight vector $\overline{W}$ can always be found by the perceptron algorithm for which the sign of $\overline{W} \cdot \overline{X}_i^T$ matches y_i for each training instance. It has been shown [421] that the perceptron algorithm always converges to provide zero error on the training data when the data are linearly separable. However, the perceptron algorithm is not guaranteed to converge in instances where the data are not linearly separable.

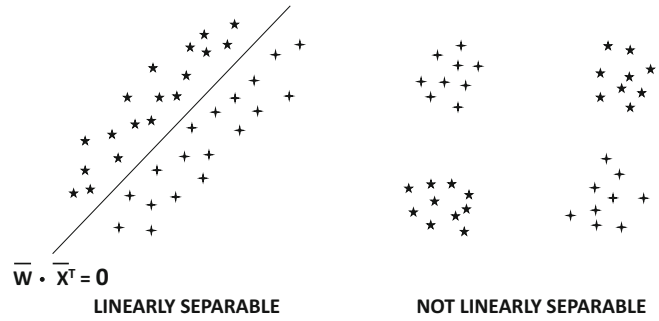


Figure 1.4: Examples of linearly separable and inseparable data in two classes

1.2.1 Use of Bias

In many settings, there is an invariant part of the prediction, which is referred to as the *bias*. For example, consider a setting in which the feature variables are mean centered, but the mean of the binary class prediction from $\{-1, +1\}$ is not 0. This will tend to occur in situations in which the binary class distribution is highly imbalanced. In such a case, the aforementioned approach is not sufficient for prediction since the different training points sum to zero, whereas the class variable do not. This is because adding up all the predictions over the training points (ignoring the sign operator), we obtain the following undesirable property:

$$\overline{W} \cdot \underbrace{\sum_i \overline{X}_i^T}_{=0} \neq \underbrace{\sum_i y_i}_{\neq 0}$$

When the differences are large, this model will perform poorly evenon the training data. We need to incorporate an additional bias variable b that captures the invariant part of the prediction:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}^T + b\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j + b\} \tag{1.4}$$

The bias can be incorporated as the weight of an edge by using a *bias neuron*. This is achieved by adding a neuron that always transmits a value of 1 to the output node. The weight of the edge connecting the bias neuron to the output node provides the bias variable. An example of a bias neuron is shown in Figure 1.3(b). Another approach that works well with single-layer architectures is to use a *feature engineering trick* in which an additional feature is created with a constant value of 1. The coefficient of this feature provides the bias, and one can then work with Equation 1.1. Throughout this book, biases will not be explicitly used (for simplicity in architectural representations) because they can be incorporated with bias neurons. However, they are often important to use in practice, as the effect on prediction accuracy can be quite large.

1.2.2 What Objective Function Is the Perceptron Optimizing?

Most machine learning algorithms (including neural networks) can be posed as loss optimization problems, wherein gradient descent updates are used to minimize the loss. The original perceptron paper by Rosenblatt [421] did not formally propose a loss function. In those years, these implementations were achieved using actual hardware circuits. The

Figure 1.5: The perceptron algorithm was originally implemented using hardware circuits. The image depicts the Mark I perceptron machine built in 1958. (Courtesy: Smithsonian Institute)

original *Mark I perceptron* was intended to be a machine rather than an algorithm, and custom-built hardware was used to create it (cf. Figure 1.5). The neural architecture was motivated by the (biological) *McCulloch-Pitts* model [332] of the neuron rather than a mathematical model. The general goal was to minimize the number of classification errors with a heuristic update process (in hardware) that changed weights in the "correct" direction whenever errors were made. This heuristic update strongly resembled gradient descent but it was not derived as a gradient-descent method. However, later work reverse engineered the loss function from the heuristic updates. This section will introduce this retrospective loss function.

Consider a training instance $(\overline{X}_i, y_i)$, where $\overline{X}_i$ is a row vector containing the features and y_i is the observed class variable. The *perceptron criterion* [41] is a loss function that penalizes the training instance when the sign of $\overline{W} \cdot \overline{X}_i^T$ is different from y_i:

$$L_i = \max\{-y_i(\overline{W} \cdot \overline{X}_i^T), 0\} \tag{1.5}$$

Furthermore, large absolute values of $\overline{W} \cdot \overline{X}_i^T$ that also do not match the sign of y_i are penalized to a greater degree. Gradient descent performs updates of $\overline{W}$ in the negative direction of the gradient of L_i (with respect to $\overline{W}$). One can use calculus to verify that the gradient of the loss is as follows:

$$\frac{\partial L_i}{\partial \overline{W}} = \left[\frac{\partial L_i}{\partial w_1}, \ldots \frac{\partial L_i}{\partial w_d}\right]^T = \begin{cases} -y_i \overline{X}_i^T & \text{if sign}\{\overline{W} \cdot \overline{X}_i^T\} \neq y_i \\ 0 & \text{otherwise} \end{cases}$$

Note the use of the matrix calculus notation $\frac{\partial L_i}{\partial \overline{W}}$, which is the derivative of a scalar with respect to a vector, and is simply the d-dimensional vector $\left[\frac{\partial L_i}{\partial w_1}, \ldots \frac{\partial L_i}{\partial w_d}\right]^T$. The negative of the vector is the direction with the fastest rate of reduction of the loss, which provides the rationale for the perceptron update:

$$\begin{aligned} \overline{W} &\Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} \\ &= \overline{W} + \alpha y_i \overline{X}_i^T \quad \text{[For misclassified instances]} \end{aligned}$$

This loss function exposes some of the weaknesses of the updates in the original algorithm; one can set $\overline{W}$ to the zero vector *irrespective of the training data set* in order to obtain the optimal loss value of 0. In spite of this fact, the perceptron updates continue to converge to a meaningful solution in *linearly separable training data sets* and a nonzero initialization of $\overline{W}$. In linearly separable data sets, a nonzero weight vector $\overline{W}$ exists in which the sign of $\overline{W} \cdot \overline{X}_i^T$ correctly matches the sign of y_i for *each and every* training instance $(\overline{X}_i, y_i)$. However, the behavior of the perceptron algorithm for data that are not linearly separable is rather arbitrary, and the resulting solution is sometimes not even a good approximate separator of the classes. The direct proportionality of the loss to the *magnitude* of the weight vector can dilute the goal of class separation by simply reducing the magnitude of the weight vector; it is possible for updates to worsen the number of misclassifications significantly while improving the loss. Because of this fact, the (vanilla) perceptron is not stable and can yield solutions of widely varying quality (for inseparable classes). Several variations of the perceptron were therefore proposed for inseparable data, and a natural approach is to always keep track of the best solution in terms of the number of misclassifications [133]. This approach of always keeping the best solution in one's "pocket" is referred to as the *pocket algorithm*.

1.3 The Base Components of Neural Architectures

The neural architecture of the perceptron algorithm uses a single output node, and the sign activation function. These aspects of neural architectures are a critical part of their design, and they may vary with the model at hand. This section discusses some of the common components of neural architectures.

1.3.1 Choice of Activation Function

The choice of activation function is a critical part of neural network design. Different types of nonlinear functions such as the *sign*, *sigmoid*, or *hyperbolic tangents* are commonly used. We have already seen the use of sign activation in the perceptron. We use the notation Φ to denote the activation function. A single-layer network with column vector $\overline{W}$ of weights and input (row) vector $\overline{X}$ would have a prediction of the following form:

$$\hat{y} = \Phi(\overline{W} \cdot \overline{X}^T) \tag{1.6}$$

The most basic activation function $\Phi(\cdot)$ is the linear activation, which is also referred to as the identity activation:

$$\Phi(v) = v$$

The linear activation function is often used in the output node, when the target is a real value.

The classical activation functions that were used early in the development of neural networks were the sign, sigmoid, and the hyperbolic tangent functions:

$$\Phi(v) = \text{sign}(v) \text{ (sign function)}$$

$$\Phi(v) = \frac{1}{1 + e^{-v}} \text{ (sigmoid function)}$$

$$\Phi(v) = \frac{e^{2v} - 1}{e^{2v} + 1} \text{ (tanh function)}$$

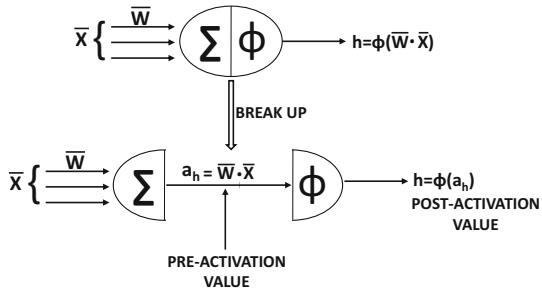


Figure 1.6: Pre- and post-activation values within a neuron

The break-up of the neuron computations into two separate values is shown in Figure 1.6. A neuron really computes two functions within the node, which is why we have incorporated the summation symbol Σ as well as the activation symbol Φ within a neuron. The value computed before applying the activation function $\Phi(\cdot)$ will be referred to as the *pre-activation value*, whereas the value computed after applying the activation function is referred to as the *post-activation value*. An important point that emerges from Figure 1.6 is that one could treat a node with a nonlinear activation as two separate computational nodes, one of which performs the linear transformation $a_h = \overline{W} \cdot \overline{X}$ and the second performs the nonlinear transformation $\Phi(a_h)$. Indeed, this type of decoupled architecture is often used in order to simplify various types of analytical results in neural networks.

While the sign activation can be used to map to binary outputs at prediction time, its non-differentiability prevents its use for creating the loss function at training time. For example, while the perceptron uses the sign function for prediction, the perceptron criterion uses only linear activation. The sigmoid activation outputs a value in $(0, 1)$, which is helpful in interpreting outputs as probabilities. The tanh function has a shape similar to that of the sigmoid function, except that it is horizontally stretched and vertically translated/re-scaled to $[-1, 1]$. The tanh and sigmoid functions are related as follows (see Exercise 3):

$$\tanh(v) = 2 \cdot \text{sigmoid}(2v) - 1$$

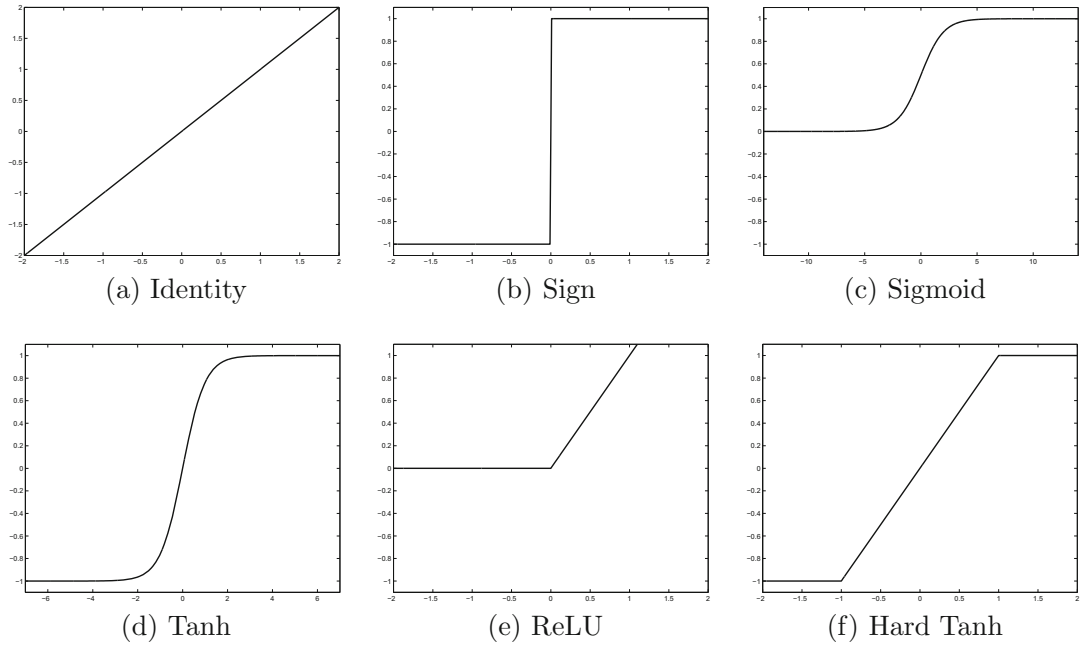
The tanh function is preferable to the sigmoid when the outputs of the computations are desired to be both positive and negative. The sigmoid and the tanh functions have been the historical tools of choice for incorporating nonlinearity in the neural network. In recent years, however, a number of piecewise linear activation functions have become more popular:

$$\Phi(v) = \max\{v, 0\} \text{ (Rectified Linear Unit [ReLU])}$$
$$\Phi(v) = \max\{\min[v, 1], -1\} \text{ (hard tanh)}$$

The ReLU and hard tanh activation functions have substantially replaced the sigmoid and soft tanh activation functions in modern neural networks (cf. Chapter 4).

Pictorial representations of all the aforementioned activation functions are illustrated in Figure 1.7. It is noteworthy that all activation functions shown here are monotonic. Furthermore, other than the identity activation function, most[1] of the other activation functions *saturate* at large absolute values of the argument at which increasing further does not change the activation much. As we will see later, such nonlinear activation functions are

[1]The ReLU shows asymmetric saturation.



(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Figure 1.7: Various activation functions

also very useful in larger and more complex neural networks, because they help in creating more powerful compositions of different types of functions. Many of these functions are referred to as *squashing* functions, as they map the outputs from an arbitrary range to bounded outputs. The use of a nonlinear activation plays a fundamental role in increasing the modeling power of a network. If a network used only linear activations, it would not provide better modeling power than a single-layer linear network. This issue is discussed in section 1.5.

1.3.2 Softmax Activation Function

The softmax activation function is unique in that it is *almost always used in the output layer* to map k real values into k probabilities of discrete events. For example, consider the k-way classification problem in which each data record needs to be mapped to one of k unordered class labels. In such cases, k output values can be used, with a *softmax activation function* with respect to k real-valued outputs $\overline{v} = [v_1, \ldots, v_k]$ at the nodes in a given layer. This activation function maps real values to probabilities that sum to 1. Specifically, the activation function for the ith output is defined as follows:

$$\Phi(\overline{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\} \tag{1.7}$$

An example of the softmax function with three outputs is illustrated in Figure 1.8, and the values v_1, v_2, and v_3 are also shown in the same figure. Note that the three outputs correspond to the probabilities of the three classes, and they convert the three outputs of the final hidden layer into probabilities with the softmax function. The final hidden layer often uses linear (identity) activations, when it is input into the softmax layer. Furthermore,

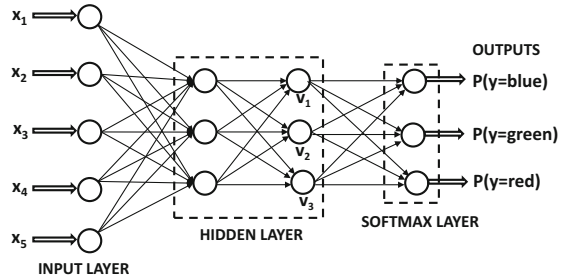


Figure 1.8: An example of multiple outputs for categorical classification with the use of a softmax layer

there are no weights associated with the softmax layer, since it is only converting real-valued outputs into probabilities. Each output is the probability of a particular class.

1.3.3 Common Loss Functions

The choice of the loss function is critical in defining the outputs in a way that is sensitive to the application at hand. For example, least-squares regression with numeric outputs requires a simple squared loss of the form $(y - \hat{y})^2$ for a single training instance with target y and prediction $\hat{y}$. For probabilistic predictions of categorical data, two types of loss functions are used, depending on whether the prediction is binary or whether it is multiway:

1. **Binary targets (logistic regression):** In this case, it is assumed that the observed value y is drawn from $\{-1, +1\}$, and the prediction $\hat{y}$ uses a sigmoid activation function to output $\hat{y} \in (0, 1)$, which indicates the probability that the observed value y is 1. Then, the negative logarithm of $|y/2 - 0.5 + \hat{y}|$ provides the loss. This is because $|y/2 - 0.5 + \hat{y}|$ indicates the probability that the prediction is correct.

2. **Categorical targets:** In this case, if $\hat{y}_1 \ldots \hat{y}_k$ are the probabilities of the k classes (using the softmax activation of Equation 1.8), and the rth class is the ground-truth class, then the loss function for a single instance is defined as follows:

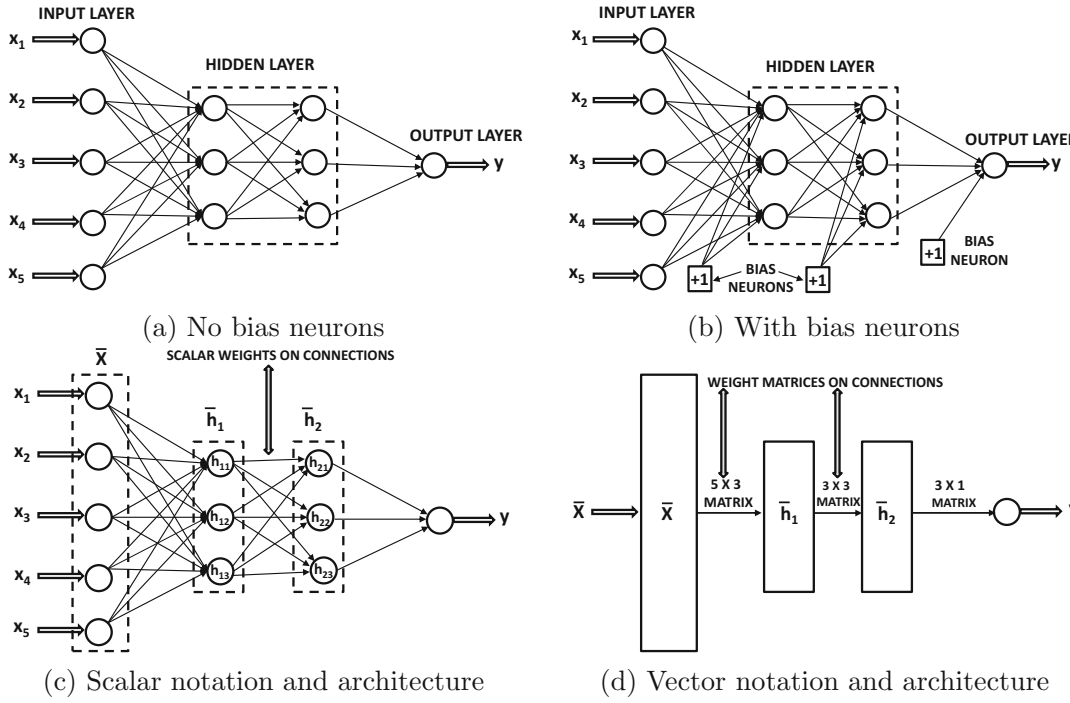
$$L = -\log(\hat{y}_r) \tag{1.8}$$

   This type of loss function implements multinomial logistic regression, and it is referred to as the *cross-entropy loss*. Note that binary logistic regression is identical to multinomial logistic regression, when the value of k is set to 2 in the latter.

The key point to remember is that the nature of the output nodes, the activation function, and the loss function depend on the application at hand.

1.4 Multilayer Neural Networks

Multilayer neural networks contain more than one computational layer. The perceptron contains an input and output layer, of which the output layer is the only computation-performing layer. The input layer transmits the data to the output layer, and all computations are completely visible to the user. Multilayer neural networks contain multiple computational layers; the additional intermediate layers (between input and output) are referred to as *hidden layers* because the computations performed are not visible to the user.



(a) No bias neurons

(b) With bias neurons

(c) Scalar notation and architecture

(d) Vector notation and architecture

Figure 1.9: The basic architecture of a feed-forward network with two hidden layers and a single output layer. Even though each unit contains a single scalar variable, one often represents all units within a single layer as a single vector unit. Vector units are often represented as rectangles and have connection *matrices* between them.

The specific architecture of multilayer neural networks is referred to as *feed-forward* networks, because successive layers feed into one another in the forward direction from input to output. The default architecture of feed-forward networks assumes that all nodes in one layer are connected to those of the next layer. Therefore, the architecture of the neural network is almost fully defined, once the number of layers and the number/type of nodes in each layer have been defined. The only remaining detail is the loss function that is optimized in the output layer. Like the perceptron criterion, the loss function penalizes undesirable deviations of predicted outputs of the neural network from the observed outputs in the training data.

As in the case of single-layer networks, bias neurons can be used both in the hidden layers and in the output layers. Examples of multilayer networks with or without the bias neurons are shown in Figures 1.9(a) and (b), respectively. In each case, the neural network contains three layers. Note that the input layer is often not counted, because it simply transmits the data and no computation is performed in that layer. If a neural network contains $p_1 \ldots p_k$ units in each of its k layers, then the (column) vector representations of these outputs, denoted by $\overline{h}_1 \ldots \overline{h}_k$ have dimensionalities $p_1 \ldots p_k$. Therefore, the number of units in each layer is referred to as the *dimensionality* of that layer.

The weights of the connections between the input layer and the first hidden layer are contained in a *matrix* W_1 with size $p_1 \times d$, whereas the weights between the rth hidden layer and the $(r+1)$th hidden layer are denoted by the $p_{r+1} \times p_r$ matrix denoted by W_r. Note

that the number of units in the $(r+1)$th layer defines the number of rows, whereas the number of units in the rth layer defines the number of columns. If the output layer contains o nodes, then the final matrix W_{k+1} of a $(k+1)$-layered network is of size $o \times p_k$. The d-dimensional input vector $\overline{x}$ is transformed into the outputs using the following recursive equations:

$$\overline{h}_1 = \Phi(W_1 \overline{x}) \qquad \text{[Input to Hidden Layer]}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1} \overline{h}_p) \quad \forall p \in \{1 \ldots k-1\} \qquad \text{[Hidden to Hidden Layer]}$$
$$\overline{o} = \Phi(W_{k+1} \overline{h}_k) \qquad \text{[Hidden to Output Layer]}$$

Although activation functions like the sigmoid function have scalar[2] arguments, they can be applied in *element-wise* fashion to their vector arguments. In other words, the activation function is applied to each of the elements of the vector to create a vector of equal length containing the outputs in the same order. One can also represent neural network architectures with vector variables. Many architectural diagrams combine the units in a single layer to create a single vector unit, which is represented as a *rectangle* rather than a *circle*. For example, the architectural diagram in Figure 1.9(c) (with scalar units) has been transformed to a vector-based neural architecture in Figure 1.9(d). Note that the weights on the connections between the vector units are now matrices. It is noteworthy that even though the *connection matrix* between the 5-dimensional input layer and the first 3-dimensional hidden layer has been annotated as a 5×3 matrix in Figure 1.9(d), one would need to use a 3×5 matrix W_1 in order to multiply the 5-dimensional column vector of inputs. *Therefore, the weight matrix dimensions are the transposes of the connection matrix dimensions.*

An implicit assumption in the vector-based neural architecture is that all units in a layer use the same activation function, which is applied in element-wise fashion to that layer. This constraint is usually not a problem, because most neural architectures use the same activation function throughout the computational pipeline, with the only deviation caused by the nature of the output layer. Throughout this book, neural architectures in which units contain vector variables will be depicted with rectangular units, whereas scalar variables will correspond to circular units.

1.4.1 The Multilayer Network as a Computational Graph

It is helpful to view a neural network as a *computational graph*, which is constructed by piecing together many basic parametric models. Neural networks are fundamentally more powerful than their building blocks because the parameters of these models are learned *jointly* to create a highly optimized composition function of these models.

A multilayer network evaluates compositions of functions computed at individual nodes. A path of length 2 in the neural network in which the function $f(\cdot)$ follows $g(\cdot)$ can be considered a composition function $f(g(\cdot))$. Furthermore, if $g_1(\cdot)$, $g_2(\cdot) \ldots g_k(\cdot)$ are the functions computed in layer m, and a particular layer-$(m+1)$ node computes $f(\cdot)$, then the composition function computed by the layer-$(m+1)$ node in terms of the layer-m inputs is $f(g_1(\cdot), \ldots g_k(\cdot))$. It is noteworthy that each of these functions is parameterized because of the presence of edge weights. Therefore, we have the following:

> A multilayer network computes a nested composition of parameterized multivariate functions. The overall function computed from the inputs to the outputs

[2]Some activation functions such as the softmax (which are typically used in the output layers) naturally have vector arguments.

can be controlled very closely by the choice of parameters. The notion of **learning** refers to the setting of the parameters to make the overall function consistent with observed input-output pairs.

The repeated nesting and the large number of nodes in each layer combine to make it difficult to even express the input-to-output function of a neural network in a compact and comprehensible way. This is a major difference from traditional machine learning algorithms in which the prediction function can often be crisply written in closed form.

The use of nonlinear activation functions is the key to increasing the power of multiple layers. If all layers use an identity activation function, then a multilayer network is no more powerful than a single-layer network (see next section). It has been shown [218] that a network with a single (nonlinear) hidden layer and a single (linear) output layer can compute almost any "reasonable" function. As a result, neural networks are often referred to as *universal function approximators*, although this theoretical claim is not always easy to translate into practical usefulness.

The weights are learned by penalizing differences between the observed and predicted output for the ith training instance with the use of a loss L_i (analogous to the perceptron criterion). The weights $\overline{W}$ of the neural network are then updated with gradient descent to reduce each loss L_i:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} \tag{1.9}$$

Similar to the perceptron, training instances are fed to the neural network one by one to make these updates.

How does one compute these gradients of the form $\frac{\partial L_i}{\partial \overline{W}}$? Multilayer neural networks (often) do not create straightforward loss functions like the perceptron that can be expressed (concisely) in closed form. The availability of closed-form loss functions in traditional machine learning is really a luxury that is helpful in using calculus to differentiate the loss function with respect to the parameters and perform updates like Equation 1.9. The fact that the loss functions of multilayer neural networks are often too awkward (and massive) to write in closed form creates challenges from the perspective of gradient descent. In this respect, the crown jewel of the neural network paradigm is the *backpropagation algorithm*; this algorithm can work out the complicated parameter update steps based on the structure of the underlying computational graph and the functions computed in each node. The analyst does not have to spend the time and effort to explicitly compute the loss function in closed form and work out the derivatives with the use of calculus. Working out these steps is often the most difficult part of most machine learning algorithms, and an important contribution of the neural network paradigm is to bring modular thinking into machine learning. In other words, the modularity in neural network design translates to modularity in learning its parameters; the specific name for the latter type of modularity is "backpropagation." This makes the design of neural networks more of an (experienced) engineer's task rather than a mathematical exercise. The backpropagation algorithm is discussed in detail in Chapter 2.

The "building block" description is particularly appropriate for multilayer neural networks. Very often, off-the-shelf softwares for building neural networks[3] provide analysts with access to these building blocks. The analyst is able to specify the number and type of units in each layer along with an off-the-shelf or customized loss function. A deep neural network containing tens of layers can often be described in a few hundred lines of code. This

[3]Examples include Torch [597], Theano [598], and TensorFlow [599].

makes the process of trying different types of architectures relatively painless for the analyst. Building a neural network with many of the off-the-shelf softwares is often compared to a child constructing a toy from building blocks that appropriately fit with one another. Each block is like a unit (or a layer of units) with a particular type of activation. All these off-the-shelf softwares have the goal of learning the weights of the neural network with the backpropagation algorithm.

1.5 The Importance of Nonlinearity

At its most basic level, a neural network is a computational graph that performs compositions of simpler functions to provide a more complex function. Much of the power of deep learning arises from the fact that the *repeated* composition of functions has significant expressive power. Not all base functions are equally good at achieving this goal. In fact, the nonlinear squashing functions used in neural networks are not arbitrarily chosen, but are carefully designed because of certain types of properties. For example, imagine a situation in which the identity activation function is used in each layer, so that only linear functions are computed. In such a case, the resulting neural network is no stronger than a single-layer, linear network:

Theorem 1.5.1 *A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network.*

Proof: Consider a network containing k hidden layers, and therefore contains a total of $(k+1)$ computational layers (including the output layer). The corresponding $(k+1)$ weight matrices between successive layers are denoted by $W_1 \ldots W_{k+1}$. Let $\overline{x}$ be the d-dimensional column vector corresponding to the input, $\overline{h}_1 \ldots \overline{h}_k$ be the column vectors corresponding to the hidden layers, and $\overline{o}$ be the m-dimensional column vector corresponding to the output. Then, we have the following recurrence condition for multi-layer networks:

$$\overline{h}_1 = \Phi(W_1\overline{x}) = W_1\overline{x}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1}\overline{h}_p) = W_{p+1}\overline{h}_p \quad \forall p \in \{1 \ldots k-1\}$$
$$\overline{o} = \Phi(W_{k+1}\overline{h}_k) = W_{k+1}\overline{h}_k$$

In all the cases above, the activation function $\Phi(\cdot)$ has been set to the identity function. Then, by eliminating the hidden layer variables, we obtain the following:

$$\overline{o} = \underbrace{W_{k+1}W_k \ldots W_1}_{W_{xo}}\overline{x}$$

Note that one can replace the matrix $W_{k+1}W_k \ldots W_1$ with the new $d \times m$ matrix W_{xo}, and learn the coefficients of W_{xo} instead of those of all the matrices W_1, $W_2 \ldots W_{k+1}$, without loss of expressivity. In other words, we have the following:

$$\overline{o} = W_{xo}\overline{x}$$

However, this condition is exactly identical to that of linear regression with multiple outputs [7]. Therefore, a multilayer neural network with identity activations does not gain over a single-layer network in terms of expressivity. ■

The aforementioned result is for the case of regression modeling with numeric target variables. A similar result holds true for binary target variables. In the special case, where

all layers use identity activation and the final layer uses a single output with sign activation for prediction, the multilayer neural network reduces to the perceptron. This result is summarized below.

Lemma 1.5.1 *Consider a multilayer network in which all hidden layers use identity activation and the single output node with linear activation that uses the perceptron criterion as the loss function. This neural network reduces to the single-layer perceptron.*

Since multilayer neural networks perform repeated compositions of functions of the form $f(g(\cdot))$, the above result can also be stated as follows:

Observation 1.5.1 *The composition of linear functions is always a linear function. The repeated composition of simple nonlinear functions can be a very complex nonlinear function.*

These observations suggest that deep networks largely make sense only when the activation functions in intermediate layers are non-linear. Typically, the functions like sigmoid and tanh are *squashing* functions in which the output is bounded within an interval, and the gradients are largest near zero values. For large absolute values of their arguments, these functions are said to reach *saturation* where increasing the absolute value of the argument further does not change its value significantly.

The universal approximation result of neural networks [218] posits that of combination of many squashing functions (like sigmoid) in a single hidden layer followed by a linear layer can be used to approximate any function well. Therefore, a two-layer network is sufficient as long as the number of hidden units is large enough. However, some kind of basic non-linearity in the activation function is always required in order to model the turns and twists in an arbitrary function. To understand this point, note that all 1-dimensional functions can be approximated as a sum of scaled/translated step functions and most of the activation functions discussed in this chapter (e.g., sigmoid) look awfully like step functions (see Figure 1.7). This basic idea is the essence of the universal approximation theorem of neural networks. However, this theoretical result cannot always be translated into practical usefulness with a limited amount of data.

1.5.1 Nonlinear Activations in Action

The previous section provides a concrete proof of the fact that a neural network with only linear activations does not gain from increasing the number of layers in it. For example, consider the two-class data set illustrated in Figure 1.10, which is represented in two dimensions denoted by x_1 and x_2. There are two instances, A and B, of the class denoted by '*' with coordinates $(1, 1)$ and $(-1, 1)$, respectively. There is also a single instance B of the class denoted by '+' with coordinates $(0, 1)$, A neural network with only linear activations will never be able to classify the training data perfectly because the points are not linearly separable.

On the other hand, consider a situation in which the hidden units have ReLU activation, and they learn the two new features h_1 and h_2, which are as follows:

$$h_1 = \max\{x_1, 0\}$$
$$h_2 = \max\{-x_1, 0\}$$

Note that these goals can be achieved by using appropriate weights from the input to hidden layer, and also applying a ReLU activation unit. The latter achieves the goal of thresholding negative values to 0. We have indicated the corresponding weights in the

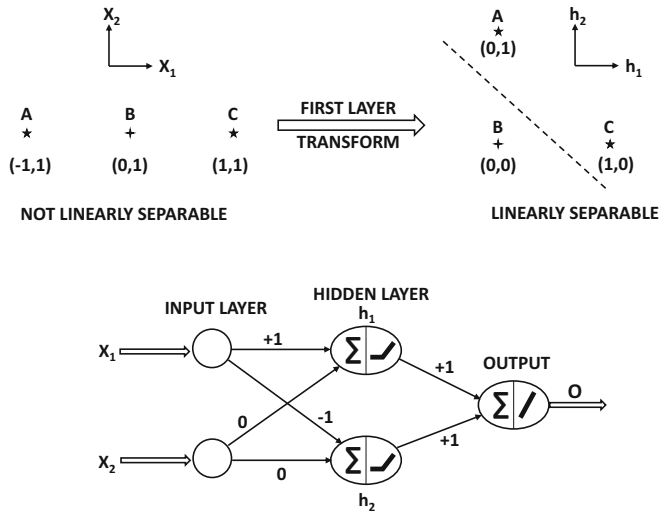


Figure 1.10: The power of nonlinear activation functions in transforming a data set to linear separability

neural network shown in Figure 1.10. We have shown a plot of the data in terms of h_1 and h_2 in the same figure. The coordinates of the three points in the 2-dimensional hidden layer are $\{(1,0),(0,1),(0,0)\}$. It is immediately evident that the two classes become linearly separable in terms of the new hidden representation. In a sense, the task of the first layer was *representation learning* to enable the solution of the problem with a linear classifier. Therefore, if we add a single linear output layer to the neural network, it will be able to classify these training instances perfectly. The key point is that the use of the nonlinear ReLU function is crucial in ensuring this linear separability. *Activation functions enable nonlinear mappings of the data, so that the embedded points can become linearly separable.* In fact, if both the weights from hidden to output layer are set to 1 with a linear activation function, the output O will be defined as follows:

$$O = h_1 + h_2 \tag{1.10}$$

This simple linear function separates the two classes because it always takes on the value of 1 for the two points labeled '*' and takes on 0 for the point labeled '+'. Therefore, much of the power of neural networks is hidden in the use of activation functions. The weights shown in Figure 1.10 will need to be *learned* in a data-driven manner, although there are many alternative choices of weights that can make the hidden representation linearly separable. Therefore, the learned weights may be different than the ones shown in Figure 1.10 if actual training is performed. Nevertheless, in the case of the perceptron, there is no choice of weights at which one could hope to classify this training data set perfectly because the data set is not linearly separable in the original space. In other words, the activation functions enable nonlinear transformations of the data, that become increasingly powerful with multiple layers. A sequence of nonlinear activations imposes a specific type of structure on the learned model, whose power increases with the depth of the sequence (i.e., number of layers in the neural network).

Another classical example is the XOR function in which the two points $\{(0,0),(1,1)\}$ belong to one class, and the other two points $\{(1,0),(0,1)\}$ belong to the other class.

It is possible to use ReLU activation to separate these two classes as well, although bias neurons will be needed in this case (see Exercise 1). The original backpropagation paper [425] discusses the XOR function, because this function was one of the motivating factors for designing multilayer networks and the ability to train them. The XOR function is considered a litmus test to determine the basic feasibility of a particular family of neural networks to properly predict nonlinearly separable classes. Although we have used the ReLU activation function above for simplicity, it is possible to use most of the other nonlinear activation functions to achieve the same goals. There are several types of neural architectures that are used commonly in various machine learning applications. The subsequent section will discuss these architectures.

1.6 Advanced Architectures and Structured Data

Numerous advanced architectures are available to address different types of data and learning settings. Some of these architectures have fallen out of favor in terms of practical application, but are nevertheless important because of the principles they embody; two such architectures are *radial basis function networks* and *restricted Boltzmann machines*. Both these architectures are closely related to classical machine learning methods. The former is a generalization of a classical machine learning method, known as the *support vector machine*, whereas the latter is a special case of a classical machine learning family referred to as *probabilistic graphical models*. Therefore, understanding these models helps in relating neural networks to classical machine learning methods. A discussion of radial basis function networks is provided in Chapter 6, and a discussion of restricted Boltzmann machine is provided in Chapter 7. Numerous other advanced topics such as *attention mechanisms*, *generative adversarial networks*, and *Kohonen self-organizing maps* are discussed in Chapter 12.

Many types of data have structure embedded in them, where the individual elements of the data are not independent of one another. For example, in a text document, successive words are not independent of one another, and in a time series, the successive elements of the series are not independent of one another. Similarly, in an image data set, adjacent pixels have values that are highly correlated with one another. In all these cases, it makes sense to use our domain-specific understanding of the data (e.g., text or image) in order to design specialized neural architectures. This book will discuss three well-known neural architectures that are specifically designed for sequence, image, and graph data.

1. *Sequence data:* Recurrent neural networks are designed for sequential data like text sentences, time-series, and other discrete sequences like biological sequences. In these cases, the input is of the form $\overline{x}_1 \ldots \overline{x}_n$, where $\overline{x}_t$ is a d-dimensional point received at the time-stamp t. For example, the vector $\overline{x}_t$ might contain the d values at the tth tick of a multivariate time-series (with d different series). In a text-setting, the vector $\overline{x}_t$ will contain the *one-hot encoded* word at the tth time-stamp. In one-hot encoding, we have a vector of length equal to the lexicon size, and the component for the relevant word has a value of 1. All other components are 0. The sequential dependencies among such inputs can be addressed with recurrent neural networks, which are discussed in Chapter 8.

2. *Image data:* The spatial nature of image data is accommodated with layers that are spatially structured. In other words, the features in the layer are spatially related to one another just like pixels in an image. Such spatial layers require the use of different

types of operations in intermediate nodes, which take the spatial arrangement of features into account. One such important operation is the *convolution operation*, from which a convolutional neural network derives its name. Convolutional neural networks are discussed in Chapter 9.

3. *Graph data:* Many real-world networked entities such as the Web and various types of social networks are represented as graphs. Graph neural networks implicitly create a neural network structure that is based on the graph that it learns it. The construction of graph neural networks is based on principles of both recurrent networks and convolutional networks. Graph neural networks are discussed in Chapter 10.

Convolutional neural networks have historically been the most successful of all types of neural networks. They are used widely for image recognition, object detection/localization, and even text processing. The performance of these networks has recently exceeded that of humans in the problem of image classification [194].

1.7 Two Notable Benchmarks

The benchmarks used in the neural network literature are dominated by data from the domain of computer vision. Although traditional machine learning data sets like the UCI repository [625] can be used for testing neural networks, the general trend is towards using data sets from perceptually oriented data domains that can be visualized well. Although there are a variety of data sets drawn from the text and image domains, two of them stand out because of their ubiquity in deep learning papers. Although both are data sets drawn from computer vision, the first of them is simple enough that it can also be used for testing generic applications beyond the field of vision. In the following, we provide a brief overview of these two data sets.

1.7.1 The MNIST Database of Handwritten Digits

The MNIST database, which stands for *Modified National Institute of Standards and Technology* database, is a large database of handwritten digits [287]. As the name suggests, this data set was created by modifying an original database of handwritten digits provided by NIST. The data set contains 60,000 training images and 10,000 testing images. Each image is a scan of a handwritten digit from 0 to 9, and the differences between different images are a result of the differences in the handwriting of different individuals. These individuals were American Census Bureau employees and American high school students. The original black and white images from NIST were size normalized to fit in a 20×20 pixel box while preserving their aspect ratio and centered in a 28×28 image by computing the center of mass of the pixels. The images were translated to position this point at the center of the 28×28 field. Each of these 28×28 pixel values takes on a value from 0 to 255, depending on where it lies in the grayscale spectrum. The labels associated with the images correspond to the ten digit values. Examples of the digits in the MNIST database are illustrated in Figure 1.11. The size of the data set is rather small, and it contains only a simple object corresponding to a digit. Therefore, one might argue that the MNIST database is a toy data set. However, its small size and simplicity is also an advantage because it can be used as a laboratory for quick testing of machine learning algorithms. Furthermore, the simplification of the data set by virtue of the fact that the digits are (roughly) centered makes it easy to use it to test algorithms beyond computer vision. Computer vision algorithms require

Figure 1.11: Examples of handwritten digits in the MNIST database

specialized assumptions such as translation invariance. The simplicity of this data set makes these assumptions unnecessary. It has been remarked by Geoff Hinton [623] that the MNIST database is used by neural network researchers in much the same way as biologists use fruit flies for early and quick results (before serious testing on more complex organisms).

Although the matrix representation of each image is suited to a convolutional neural network, one can also convert it into a multidimensional representation of $28 \times 28 = 784$ dimensions. This conversion loses some of the spatial information in the image, but this loss is not debilitating (at least in the case of the MNIST data set) because of its relative simplicity. In fact, the use of a simple support vector machine on the 784-dimensional representation can provide an impressive error rate of about 0.56%. A straightforward 2-layer neural network on the multidimensional representation (without using the spatial structure in the image) generally does worse than the support vector machine across a broad range of parameter choices! A deep neural network without any special convolutional architecture can achieve an error rate of 0.35% [75]. Deeper neural networks and convolutional neural networks (that do use spatial structure) can reduce the error rate to as low as 0.21% by using an ensemble of five convolutional networks [419]. Therefore, even on this simple data set, one can see that the relative performance of neural networks with respect to traditional machine learning is sensitive to the specific architecture used in the former.

Finally, it should be noted that the 784-dimensional non-spatial representation of the MNIST data is used for testing all types of neural network algorithms beyond the domain of computer vision. Even though the use of the 784-dimensional (flattened) representation is not appropriate for a vision task, it is still useful for testing the general effectiveness of non-vision oriented (i.e., generic) neural network algorithms. For example, the MNIST data is frequently used to test data reconstruction algorithms for multidimensional data and not just image data. Even when an MNIST image is reconstructed using a tabular (i.e., not image domain-specific) algorithm, one can visualize the reconstructed pixels to obtain a feel of what the algorithm is doing with the data. This visual exploration often gives the researcher some insights that are not available with arbitrary data sets like those obtained from the UCI Machine Learning Repository [625]. In this sense, the MNIST data set tends to have broader usability than many other types of data sets.

1.7.2 The ImageNet Database

The *ImageNet* database [605] is a huge database of over 14 million images drawn from 1000 different categories. Its class coverage is exhaustive enough that it covers most types of images that one would encounter in everyday life. This database is organized according to

a *WordNet* hierarchy of nouns [341]. The *WordNet* database is a data set containing the relationships among English words using the notion of *synsets*. The *WordNet* hierarchy has been successfully used for machine learning in the natural language domain, and therefore it is natural to design an image data set around these relationships.

The *ImageNet* database is famous for the fact that an annual *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* [606] is held using this dataset. This competition has a very high profile in the vision community and receives entries from most major research groups in computer vision. The entries to this competition have resulted in many of the state-of-the-art image recognition architectures today, including the methods that have surpassed human performance on some narrow tasks like image classification [194]. Because of the wide availability of known results on these data sets, it is a popular alternative for benchmarking. The data set is large and diverse enough to be representative of the key visual concepts within the image domain. As a result, convolutional neural networks are often trained on this data set; the pretrained network can be used to extract features from an arbitrary image. This image representation is defined by the hidden activations in the penultimate layer of the neural network. Such an approach creates new multidimensional representations of image data sets that are amenable for use with traditional machine learning methods. One can view this approach as a kind of transfer learning in which the visual concepts in the *ImageNet* data set are transferred to unseen data objects for other applications.

1.8 Summary

Although a neural network can be viewed as a simulation of the learning process in living organisms, a more direct understanding of neural networks is as computational graphs. Such computational graphs perform recursive composition of simpler functions in order to learn more complex functions. Since these computational graphs are parameterized, the problem generally boils down to learning the parameters of the graph in order to optimize a loss function. The simplest types of neural networks are often basic machine learning models like least-squares regression. The real power of neural networks is unleashed by using more complex combinations of the underlying functions. The main advantage of the neural paradigm is that one can use the backpropagation algorithm to compute gradients. The design of deep learning methods in specific domains such as text and images requires carefully crafted architectures, such as recurrent neural networks and convolutional neural networks.

1.9 Bibliographic Notes and Software Resources

A proper understanding of neural network design requires a solid understanding of machine learning algorithms, and especially the linear models based on gradient descent. The reader is recommended to refer to [2, 3, 40, 187] for basic knowledge on machine learning methods. Numerous surveys and overviews of neural networks in different contexts may be found in [28, 29, 208, 283, 356, 450]. Classical books on neural networks for pattern recognition may be found in [41, 192], whereas more recent perspectives on deep learning may be found in [154]. A recent text mining book [7] also discusses recent advances in deep learning for text analytics. An overview of the relationships between deep learning and computational neuroscience may be found in [186, 249].

The perceptron algorithm was proposed by Rosenblatt [421]. To address the issue of stability, the *pocket algorithm* [133], the *Maxover* algorithm [542], and other margin-based

methods [128]. Other early algorithms of a similar nature included the Widrow-Hoff [550] and the Winnow algorithms [255]. The Winnow algorithm uses multiplicative updates instead of additive updates, and is particularly useful when many features are irrelevant. The original idea of backpropagation was based on the idea of differentiation of composition of functions as developed in control theory [54, 247]. The use of dynamic programming to perform gradient-based optimization of variables that are related via a directed acyclic graph has been a standard practice since the sixties. However, the ability to use these methods for neural network training had not yet been observed at the time. In 1969, Minsky and Papert published a book on perceptrons [342], which was largely negative about the potential of being able to properly train multilayer neural networks. The book showed that a single perceptron had limited expressiveness, and no one knew how to train multiple layers of perceptrons anyway. Minsky was an influential figure in artificial intelligence, and the negative tone of his book contributed to the first winter in the field of neural networks. The adaptation of dynamic programming methods to backpropagation in neural networks was first proposed by Paul Werbos in his PhD thesis in 1974 [543]. However, Werbos's work could not overcome the strong views against neural networks that had already become entrenched at the time. The backpropagation algorithm was proposed again by Rumelhart *et al.* in 1986 [424, 425]. Rumelhart *et al.*'s work is significant for the beauty of its presentation, and it was able to address at least some of the concerns raised earlier by Minsky and Papert. This is one of the reasons that the Rumelhart *et al.* paper is considered very influential from the perspective of backpropagation, even though it was certainly not the first to propose the method. A discussion of the history of the backpropagation algorithm may be found in the book by Paul Werbos [544].

At this point, the field of neural networks was only partially resurrected, as there were still problems with training neural networks. Furthermore, backpropagation turned out to be less effective at training deeper networks because of the vanishing and exploding gradient problems. However, by this time, it was already hypothesized by several prominent researchers that existing algorithms would yield large performance improvements with increases in data, computational power, and algorithmic experimentation. The coupling of big data frameworks with GPUs turned out to be a boon for neural network research in the late 2000s. With reduced cycle times for experimentation enabled by increased computational power, tricks like pretraining started showing up in the late 2000s [208]. The publicly obvious resurrection of neural networks occurred after the year 2011 with the resounding victories [263] of neural networks in deep learning competitions for image classification. The consistent victories of deep learning algorithms in these competitions laid the foundation for the explosion in popularity we see today. Notably, the differences of these winning architectures from the ones that were developed more than two decades earlier are modest (but essential).

The theoretical expressiveness of neural networks was recognized early in its development. For example, early work recognized that a neural network with a single hidden layer can be used to approximate any function [218]. A further result is that certain neural architectures like recurrent networks are Turing complete [464]. The latter means that neural networks can potentially simulate any algorithm. Of course, there are numerous practical issues associated with neural network training, as to why these exciting theoretical results do not always translate into real-world performance.

Video Lectures

Deep learning has a significant number of free video lectures available on resources such as *YouTube* and *Coursera*. Two of the most authoritative resources include Geoff Hinton's course on YouTube [623] and Andrew Ng's course at *Coursera* [624]. *Coursera* has multiple offerings on deep learning, and offers a group of related courses in the area. During the writing of this book, an accessible course from Andrew Ng was also added to the offerings. A course on convolutional neural networks from Stanford University is freely available on *YouTube* [246]. The Stanford class by Karpathy, Johnson, and Fei-Fei [246] is on convolutional neural networks, although it does an excellent job in covering broader topics in neural networks. The initial parts of the course deal with vanilla neural networks and training methods.

Numerous topics in machine learning [92] and deep learning [93] are covered by Nando de Freitas in a lectures available on *YouTube*. Another interesting class on neural networks is available from Hugo Larochelle at the Universite de Sherbrooke [268]. A deep learning course by Ali Ghodsi at the University of Waterloo is available at [143]. David Silver's course on reinforcement learning is available at [641].

Software Resources

Deep learning is supported by numerous software frameworks like *Caffe* [596], *Torch* [597], *Theano* [598], and *TensorFlow* [599]. Extensions of *Caffe* to Python and MATLAB are available. *Caffe* was developed at the University of California at Berkeley, and it is written in C++. It provides a high-level interface in which one can specify the architecture of the network, and it enables the construction of neural networks with very little code writing and relatively simple scripting. The main drawback of *Caffe* is the relatively limited documentation available. *Theano* [35] is Python-based, and it provides high-level packages like *Keras* [600] and *Lasagne* [601] as interfaces. *Theano* is based on the notion of computational graphs, and most of the capabilities provided around it use this abstraction explicitly. *TensorFlow* [599] is also strongly oriented towards computational graphs, and is the framework proposed by Google. *Torch* [597] is written in a high-level language called *Lua*, and it is relatively friendly to use. In recent years, *Torch* has gained some ground compared to other frameworks. Support for GPUs is tightly integrated in *Torch*, which makes it relatively easy to deploy *Torch*-based applications on GPUs. Many of these frameworks contain pretrained models from computer vision and text mining, which can be used to extract features. Many off-the-shelf tools for deep learning are available from the **DeepLearning4j** repository [614]. IBM has a *PowerAI* platform that offers many machine learning and deep learning frameworks on top of IBM Power Systems [622]. Notably, as of the writing of this book, this platform also has a free edition available for certain uses.

1.10 Exercises

1. Consider the case of the XOR function in which the two points $\{(0,0),(1,1)\}$ belong to one class, and the other two points $\{(1,0),(0,1)\}$ belong to the other class. Show how you can use the ReLU activation function to separate the two classes in a manner similar to the example in Figure 1.10.

2. Show the following properties of the sigmoid and tanh activation functions (denoted by $\Phi(\cdot)$ in each case):

(a) Sigmoid activation: $\Phi(-v) = 1 - \Phi(v)$

(b) Tanh activation: $\Phi(-v) = -\Phi(v)$

(c) Hard tanh activation: $\Phi(-v) = -\Phi(v)$

3. Show that the tanh function is a re-scaled sigmoid function with both horizontal and vertical stretching, as well as vertical translation:

$$\tanh(v) = 2\text{sigmoid}(2v) - 1$$

4. Consider a data set in which the two points $\{(-1, -1), (1, 1)\}$ belong to one class, and the other two points $\{(1, -1), (-1, 1)\}$ belong to the other class. Start with perceptron parameter values at $(0, 0)$, and work out a few point-wise gradient-descent updates with $\alpha = 1$. While performing the gradient-descent updates, cycle through the training points in any order.

(a) Does the algorithm converge in the sense that the change in objective function becomes extremely small over time?

(b) Explain why the situation in (a) occurs.

5. For the data set in Exercise 4, where the two features are denoted by (x_1, x_2), define a new 1-dimensional representation z denoted by the following:

$$z = x_1 \cdot x_2$$

Is the data set linearly separable in terms of the 1-dimensional representation corresponding to z? Explain the importance of nonlinear transformations in classification problems.

6. Implement the perceptron in a programming language of your choice.

7. Show that the derivative of the sigmoid activation function is at most 0.25, irrespective of the value of its argument. At what value of its argument does the sigmoid activation function take on its maximum value?

8. Show that the derivative of the tanh activation function is at most 1, irrespective of the value of its argument. At what value of its argument does the tanh activation take on its maximum value?

9. Consider a network with two inputs x_1 and x_2. It has two hidden layers, each of which contain two units. Assume that the weights in each layer are set so that top unit in each layer applies sigmoid activation to the sum of its inputs and the bottom unit in each layer applies tanh activation to the sum of its inputs. Finally, the single output node applies ReLU activation to the sum of its two inputs. Write the output of this neural network *in closed form* as a function of x_1 and x_2. This exercise should give you an idea of the complexity of functions computed by neural networks.

10. Compute the partial derivative of the closed form computed in the previous exercise with respect to x_1. Is it practical to compute derivatives for gradient descent in neural networks by using closed-form expressions (as in traditional machine learning)?

11. Consider a 2-dimensional data set in which all points with $x_1 > x_2$ belong to the positive class, and all points with $x_1 \leq x_2$ belong to the negative class. Therefore, the true separator of the two classes is linear hyperplane (line) defined by $x_1 - x_2 = 0$. Now create a training data set with 20 points randomly generated inside the unit square in the positive quadrant. Label each point depending on whether or not the first coordinate x_1 is greater than its second coordinate x_2. Implement the perceptron algorithm, train it on the 20 points above, and test its accuracy on 1000 randomly generated points inside the unit square. Generate the test points using the same procedure as the training points.

12. Suppose you know (as domain knowledge) that the real-valued output of a neural model *always* lies in the range $[a, b]$. Show how you can use an activation function in the output or hidden layer(s) to model this fact.

Chapter 2

The Backpropagation Algorithm

"Don't re-invent the wheel, just re-align it." – Anthony J. D'Angelo

2.1 Introduction

This chapter will introduce the backpropagation algorithm, which is the key to learning in multilayer neural networks. In the early years, methods for training multilayer networks were not known, primarily because of the unfamiliarity of the computer science community with ideas that were used quite frequently in control theory [54, 247]. In their influential book, Minsky and Papert [342] strongly argued against the prospects of neural networks because of the (perceived) inability to train multilayer networks. Therefore, neural networks stayed out of favor as a general area of research till the eighties. At this point, an algorithm for training neural networks was popularized[1] by Rumelhart *et al.* [424, 425] in the form of the "backpropagation" algorithm. This algorithm had its roots in control theory, and was re-invented several times [54, 247, 543, 544]. The introduction of this algorithm rekindled an interest in neural networks. However, several computational, stability, and overfitting challenges were found in the use of this algorithm. As a result, research in the field of neural networks again fell from favor.

At the turn of the century, several advances again brought popularity to neural networks. Not all of these advances were algorithm-centric. For example, increased data availability and computational power have played the primary role in this resurrection. However, some changes to the basic backpropagation algorithm and clever methods for initialization have also helped. It has also become easier in recent years to perform the intensive experimentation required for making algorithmic adjustments due to the reduced testing cycle times

[1] Although the backpropagation algorithm was popularized by the Rumelhart *et al.* papers [424, 425], it had (independently) been studied earlier in the context of control theory. Crucially, Paul Werbos's forgotten (and eventually rediscovered) thesis [543] in 1974 discussed how these backpropagation methods could be used in neural networks. Nevertheless, Rumelhart *et al.*'s papers in 1986 were significant in contributing to a better understanding and appreciation of these ideas.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_2

(caused by improved computational hardware). Therefore, increased data, computational power, and reduced experimentation time (for algorithmic tweaking) went hand-in-hand.

Chapter Organization

This chapter is organized as follows. The computational graph abstraction for neural networks is introduced in section 2.2. Section 2.3 discusses the backpropagation algorithm from the perspective of computational graphs. The generalization of the computational graph abstraction to neural networks is discussed in section 2.4. The vector-centric view of backpropagation is introduced in section 2.5. Important details and special cases are discussed in section 2.6. Feature preprocessing and initialization issues are discussed in section 2.7. The interpretability of backpropagation is presented in section 2.8. The summary is presented in section 2.9.

2.2 The Computational Graph Abstraction

A *computational graph* is a more general abstraction than a neural network, and it is defined as follows:

Definition 2.2.1 (Directed Acyclic Computational Graph) *A directed acyclic computational graph is a directed acyclic graph of nodes, where each node contains a variable. Edges* **might** *be associated with learnable parameters. A variable in a node is either fixed externally (for input nodes with no incoming edges), or it is a computed as a function of the variables in the tail ends of edges incoming into the node and the learnable parameters on the incoming edges.*

Note that the above definition of computational graphs does not make any assumptions on the type of function computed in a node, although it is common to use a combination of a linear function and an activation function in a neural network. Furthermore, the nodes of the computational graph need not be arranged in layers, as is common in the case of neural networks. The main restriction is that the graph needs to be acyclic in order for computations of variables to be performed in a clearly defined way.

The variables in some of the nodes are observable, and are referred to as *output nodes*. These output nodes are associated with loss functions that are computed using observed input-output pairs in the training data. For each observed input, the loss function quantifies how well the predicted values in output nodes match the observed values of the output (when input node variables are fixed to the observed inputs). The choice of loss function depends on the application at hand. For example, one can model least-squares regression by using as many input nodes as the number of input variables (regressors), and a single output node containing the predicted regressand. In this model, directed edges exist from each input node to the solitary output node, and the parameter on each such edge corresponds to the weight w_i associated with that input variable x_i (cf. Figure 2.1). The output node, which is the only computational node, computes the following function of the variables $x_1 \ldots x_d$ in the d input nodes:

$$\hat{o} = f(x_1, x_2, \ldots, x_d) = \sum_{i=1}^{d} w_i x_i$$

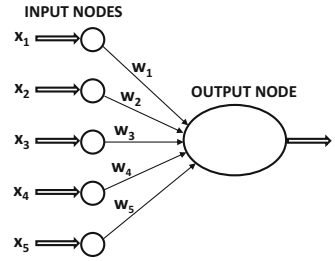


Figure 2.1: A single-layer computational graph that can perform linear regression

The computational graph in Figure 2.1 is a rather rudimentary one in which the computed function is not particularly complex. However, the closed-form representation of the function computed by a neural network becomes increasingly unwieldy and complex as one moves from a single-layer network to multiple layers.

2.2.1 Computational Graphs Create Complex Functions

Any computational graph (such as a neural network) evaluates compositions of functions computed at individual nodes. A path of length 2 in a computational graph in which the function $f(\cdot)$ follows $g(\cdot)$ can be considered a composition function $f(g(\cdot))$. It is this type of *recursive nesting* of functions that makes it hard to express the output (or loss function) of a computational graph in closed form as a *direct* function of its *inputs* and *edge-specific parameters*. The inability to easily express the optimization function in closed form in terms of the edge-specific parameters (as is common in all machine learning problems) causes difficulties in computing the derivatives needed for gradient descent.

Consider a variable x at a node in a computational graph with only three nodes containing a path of length 2. The first node applies the function $g(x)$, whereas the second node applies the function $f(\cdot)$ to the result. Such a graph computes the function $f(g(x))$, and it is shown in Figure 2.2. The example shown in Figure 2.2 uses the case when $f(x) = \cos(x)$ and $g(x) = x^2$. Therefore, the overall function is $\cos(x^2)$. Now, consider another setting in which both $f(x)$ and $g(x)$ are set to the same function, which is the sigmoid function:

$$f(x) = g(x) = \frac{1}{1 + \exp(-x)}$$

Then, the global function evaluated by the computational graph is as follows:

$$f(g(x)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-x)}\right]} \tag{2.1}$$

This simple graph already computes a rather awkward composition function. Trying to find the derivative of this composition function algebraically becomes increasingly tedious as we increase the complexity of the computational graph.

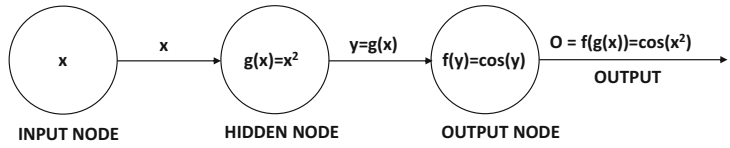


Figure 2.2: A simple computational graph with an input node and two computational nodes

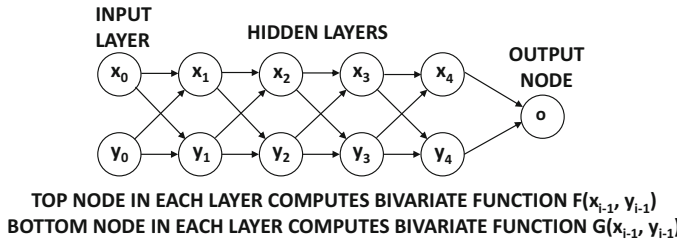


Figure 2.3: The awkwardness of recursive nesting caused by a computational graph

Consider a case in which the functions $g_1(\cdot)$, $g_2(\cdot) \ldots g_k(\cdot)$ are the functions computed in layer m, and they feed into a particular layer-$(m+1)$ node that computes the multivariate function $f(\cdot)$ that uses the values computed in the previous layer as arguments. Therefore, the layer-$(m+1)$ function computes $f(g_1(\cdot), \ldots g_k(\cdot))$. This type of multivariate composition function already appears rather awkward. As we increase the number of layers, a function that is computed several edges downstream will have as many layers of nesting as the length of the path from the source to the final output. For example, if we have a computational graph which has 10 layers, and 2 nodes per layer, the overall composition function would have 2^{10} nested "terms" if one were to try to write it on a piece of paper (in closed form). In fact, such a "closed-form" function will not even fit on any reasonably sized sheet of paper that we are accustomed to writing on!

In order to understand this point, consider the function in Figure 2.3. In this case, we have two nodes in each layer other than the output layer. The output layer simply sums its inputs. Each hidden layer contains two nodes. The variables in the ith layer are denoted by x_i and y_i, respectively. The input nodes (variables) use subscript 0, and therefore they are denoted by x_0 and y_0 in Figure 2.3. The two computed functions in the ith layer are $F(x_{i-1}, y_{i-1})$ and $G(x_{i-1}, y_{i-1})$, respectively.

In the following, we will write the expression for the variable in each node in order to show the increasing complexity with increasing number of layers:

$$x_1 = F(x_0, y_0)$$
$$y_1 = G(x_0, y_0)$$
$$x_2 = F(x_1, y_1) = F(F(x_0, y_0), G(x_0, y_0))$$
$$y_2 = G(x_1, y_1) = G(F(x_0, y_0), G(x_0, y_0))$$

We can already see that the expressions have already started looking unwieldy. On computing the values in the next layer, this becomes even more obvious:

$$x_3 = F(x_2, y_2) = F(F(F(x_0, y_0), G(x_0, y_0)), G(F(x_0, y_0), G(x_0, y_0)))$$
$$y_3 = G(x_2, y_2) = G(F(F(x_0, y_0), G(x_0, y_0)), G(F(x_0, y_0), G(x_0, y_0)))$$

An immediate observation is that the complexity and length of the closed-form function increases *exponentially* with the path lengths in the computational graphs. This type of complexity further increases in the case when optimization parameters are associated with the edges, and one tries to express the outputs/losses in terms of the inputs and the parameters on the edges. This is obviously a problem, if we try to use the standard approach of first expressing each variable in closed form in terms of the optimization parameters on the edges in order to differentiate the closed-form expression. Here, a key point is that the derivatives of the output with respect to various variables in the computational graph are related to one another with the use of the chain rule of differential calculus. Therefore, the chain rule of differential calculus needs to be applied repeatedly to update derivatives of the output with respect to the variables in the computational graph. This approach is referred to as the backpropagation algorithm, because the derivatives of the output with respect to the variables close to the output are simpler to compute (and are therefore computed first while propagating them backwards towards the inputs). Furthermore, derivatives are computed *numerically* for the specific values of each input-output pair in a training point, rather than computing them *algebraically* than substituting the values of the features.

2.3 Backpropagation in Computational Graphs

To learn the weights of a computational graph, an input-output pair is selected from the training data and the variables in the input nodes are instantiated with the corresponding input variables in the training data. Then, the computations are performed in the *forward direction* on the directed acyclic graph to determine the *predicted values* of the outputs. If these *predicted values* are different from the *observed values* in the training data, a loss function is used to quantify the error. The gradients of the loss function are computed with respect to the weights in the computational graph. These gradients are used to update the weights. Therefore, the overall approach for training a computational graph is as follows:

1. Use the attribute values from the input portion of a training data point to fix the values in the input nodes. Repeatedly select a node for which the values in all incoming nodes have already been computed and apply the node-specific function to also compute its variable. Such a node can be found in a directed acyclic graph by processing the nodes in order of increasing distance from input nodes. Repeat the process until the values in all nodes (including the output nodes) have been computed. If the values on the output nodes do not match the observed values of the output in the training point, compute the loss value. This phase is referred to as the *forward phase*.

2. Compute the gradient of the loss with respect to the weights on the edges. This phase is referred to as the *backwards phase*. The rationale for calling it a "backwards phase" is that derivatives of the loss with respect to weights near the output (where the loss function is computed) are easier to compute and are computed first. The derivatives become increasingly complex as we move towards edge weights away from the output (in the backwards direction) and the chain rule is used repeatedly to compute them.

3. Update the weights in the negative direction of the gradient.

One cycles through the training points repeatedly until convergence is reached. A single cycle through all the training points is referred to as an *epoch*.

Although we want to compute the gradient of the loss function with respect to the *weights* in a computational graph. it turns out that *the derivatives of the node variables*

with respect to one another can be easily used to compute the derivative of the loss function with respect to the weights on the edges. Therefore, in this discussion, we will focus on the computation of the derivatives of the variables with respect to one another. Later, we will show how these derivatives can be converted into gradients of loss functions with respect to weights.

2.3.1 Computing Node-to-Node Derivatives with the Chain Rule

As discussed in an earlier section, one can express the function in a computational graph in terms of the nodes in early layers using an awkward closed-form expression that uses nested compositions of functions. If one were to indeed compute the derivative of this closed-form expression, it would require the use of the chain rule of differential calculus in order to deal with the repeated composition of functions. However, a blind application of the chain rule is rather wasteful in this case because many of the expressions in different portions of the inner nesting are identical, and one would be repeatedly computing the same derivative. The key idea in *automatic differentiation over computational graphs* is to recognize the fact that structure of the computational graph already provides all the information about which terms are repeated. We can avoid repeating the differentiation of these terms by using the structure of the computational graph itself to store intermediate results (by working backwards starting from output nodes to compute derivatives)! This is a well-known idea from dynamic programming, which has been used frequently in control theory [54, 247]. In neural networks, this same algorithm is referred to as *backpropagation*. It is noteworthy that the applications of this idea in control theory were well-known to the traditional optimization community in 1960 [54, 247], although they remained unknown to researchers in the field of artificial intelligence for a while (who coined the term "backpropagation" in the 1980s to independently propose and describe this idea in the context of neural networks).

The simplest version of the chain rule is defined for a univariate composition of functions:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x} \tag{2.2}$$

This variant is referred to as the *univariate chain rule*. Note that each term on the right-hand side is a *local gradient* because it computes the derivative of a *local* function with respect to its immediate argument rather than a recursively derived argument. The basic idea is that a composition of functions is applied on the input x to yield the final output, and the gradient of the final output is given by the product of the local gradients along that path. Each local gradient only needs to worry about its specific input and output, which simplifies the computation. An example is shown in Figure 2.2 in which the function $f(y)$ is $\cos(y)$ and $g(x) = x^2$. Therefore, the composition function is $\cos(x^2)$. On using the univariate chain rule, we obtain the following:

$$\frac{\partial f(g(x))}{\partial x} = \underbrace{\frac{\partial f(g(x))}{\partial g(x)}}_{-\sin(g(x))} \cdot \underbrace{\frac{\partial g(x)}{\partial x}}_{2x} = -2x \cdot \sin(x^2)$$

The example of Figure 2.2 is a rather simple case in which the computational graph is a single path. In general, a computational graph with good expressive power will not be a single path. In neural networks, a single node feeds its output to multiple nodes. This case is also a challenging one from the perspective of computing derivatives, because the closed form of the global function with respect to variables in earlier nodes becomes rather

large and unwieldy. Consider the simple case in which we have a single input x, and we have k independent computational nodes that compute the functions $g_1(x), g_2(x), \ldots g_k(x)$. If these nodes are connected to a single output node computing the function $f()$ with k arguments, then the resulting function that is computed is $f(g_1(x), \ldots g_k(x))$. In such cases, the *multivariate chain rule* needs to be used. The multivariate chain rule is defined as follows:

$$\frac{\partial f(g_1(x), \ldots g_k(x))}{\partial x} = \sum_{i=1}^{k} \frac{\partial f(g_1(x), \ldots g_k(x))}{\partial g_i(x)} \cdot \frac{\partial g_i(x)}{\partial x} \tag{2.3}$$

It is easy to see that the multivariate chain rule of Equation 2.3 is a simple generalization of that in Equation 2.2.

One can also view the multivariate chain rule in a path-centric fashion rather than a node-centric fashion. *For any pair of source-sink nodes, the multivariate chain rule can be recursively applied to derive the fact that the derivative of the variable in the sink node with respect to the variable in the source node is simply the sum of the expressions arising from the univariate chain rule being applied to all paths existing between that pair of nodes.* While this view leads to a direct expression for the derivative between any pair of nodes, it leads to an excessive amount of repeated computation. This is because the number of paths between a pair of nodes is exponentially related to the path length. In order to show the repetitive nature of the operations, we work with a very simple closed-form function with a single input x:

$$o = \sin(x^2) + \cos(x^2) \tag{2.4}$$

The resulting computation graph is shown in Figure 2.4. In this case, the multivariate chain rule is applied to compute the derivative of the output o with respect to x. This is achieved by applying the univariate chain rule to each of the two paths from x to o and summing the expressions:

$$\begin{aligned}
\frac{\partial o}{\partial x} &= \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{-\sin(y)} \cdot \underbrace{f'(x)}_{2x} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(x)}_{2x} \\
&= -2x \cdot \sin(y) + 2x \cdot \cos(z) \\
&= -2x \cdot \sin(x^2) + 2x \cdot \cos(x^2)
\end{aligned}$$

It is easy to see that the application of the chain rule on the computational graph correctly evaluates the derivative, which is $-2x \cdot \sin(x^2) + 2x \cdot \cos(x^2)$. In this simple example, there

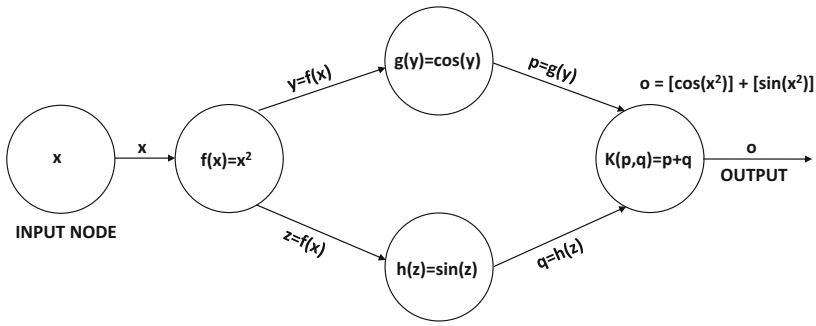


Figure 2.4: A simple computational function that illustrates the chain rule.

are two paths, both of which compute the function $f(x) = x^2$. As a result, the same function is differentiated *twice*, once for each path. Furthermore, the global gradient of o with respect to x is obtained by computing the product of the local gradients along each of the two paths and then adding them. This type of repetition can have severe effects for large multilayer networks containing many shared nodes, where the same function might be differentiated hundreds of thousands of times as a portion of the nested recursion. It is this *repeated and wasteful* approach to the computation of the derivative, that it is impractical to express the global function of a computational graph in closed form and explicitly differentiate it.

One can summarize the path-centric view of the multivariate chain rule as follows:

Lemma 2.3.1 (Pathwise Aggregation Lemma) *Consider a directed acyclic computational graph in which the ith node contains variable $y(i)$. The local derivative $z(i, j)$ of the directed edge (i, j) in the graph is defined as $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$. Let a non-null set of paths $\mathcal{P}$ exist from a node s in the graph to node t. Then, the value of $\frac{\partial y(t)}{\partial y(s)}$ is given by computing the product of the local gradients along each path in $\mathcal{P}$, and summing these products over all paths in $\mathcal{P}$.*

$$\frac{\partial y(t)}{\partial y(s)} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j) \tag{2.5}$$

This lemma can be easily shown by applying the multivariate chain rule (Equation 2.3) recursively over the computational graph. Although the use of the path-wise aggregation lemma is a wasteful approach for computing the derivative of $y(t)$ with respect to $y(s)$, it helps us develop an exponential-time algorithm for computing the derivative, which is relatively simple to understand.

An Exponential-Time Algorithm

The pathwise aggregation lemma provides a natural exponential-time algorithm, which is roughly[2] similar to the steps one would go through by expressing the computational function in closed form with respect to a particular variable and then differentiating it. Specifically, the pathwise aggregation lemma leads to the following exponential-time algorithm to compute the derivative of the output o with respect to a variable x in the graph:

1. Use computational graph to compute the value $y(i)$ of each node i in a forward phase.
2. Compute the local partial derivatives $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge in the computational graph.
3. Let $\mathcal{P}$ be the set of all paths from an input node with value x to the output o. For each path $P \in \mathcal{P}$ compute the product of each local derivative $z(i, j)$ on that path.
4. Add up these values over all paths in $\mathcal{P}$.

In general, a computational graph will have an exponentially increasing number of paths with depth and one must add the product of the local derivatives over all paths. An example is shown in Figure 2.5, in which we have five layers, each of which has only two units.

[2]It is not exactly similar in cases where the composition of functions in the computational graph leads to a simplified closed form. However, if one cannot simplify or "collapse" the composition of functions in the computational graph to a simplified form, the number of steps would be similar in the two cases.

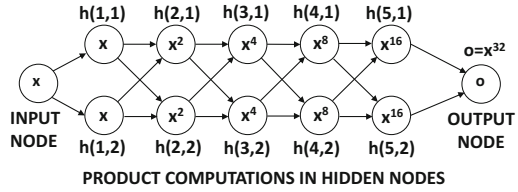


Figure 2.5: The number of paths in a computational graph increases exponentially with depth. In this case, the chain rule will aggregate the product of local derivatives along $2^5 = 32$ paths.

Therefore, the number of paths between the input and output is $2^5 = 32$. The jth hidden unit of the ith layer is denoted by $h(i, j)$. Each hidden unit is defined as the product of its inputs:

$$h(i,j) = h(i-1,1) \cdot h(i-1,2) \quad \forall j \in \{1,2\} \tag{2.6}$$

In this case, the output is x^{32}, which is expressible in closed form, and can be differentiated easily with respect to x. In other words, we do not really need computational graphs in order to perform the differentiation. However, we will use the exponential-time algorithm to elucidate its workings. The derivatives of each $h(i, j)$ with respect to its two inputs are the values of the complementary inputs, because the partial derivative of the multiplication of two variables is the complementary variable:

$$\frac{\partial h(i,j)}{\partial h(i-1,1)} = h(i-1,2), \quad \frac{\partial h(i,j)}{\partial h(i-1,2)} = h(i-1,1)$$

The pathwise aggregation lemma implies that the value of $\frac{\partial o}{\partial x}$ is the product of the local derivatives (which are the complementary input values in this particular case) along all 32 paths from the input to the output:

$$\begin{aligned}\frac{\partial o}{\partial x} &= \sum_{j_1,j_2,j_3,j_4,j_5 \in \{1,2\}^5} \underbrace{\prod h(1,j_1)}_{x} \underbrace{h(2,j_2)}_{x^2} \underbrace{h(3,j_3)}_{x^4} \underbrace{h(4,j_4)}_{x^8} \underbrace{h(5,j_5)}_{x^{16}} \\ &= \sum_{\text{All 32 paths}} x^{31} = 32x^{31}\end{aligned}$$

This result is, of course, consistent with what one would obtain on differentiating x^{32} directly with respect to x. However, an important observation is that it requires 2^5 aggregations to compute the derivative in this way for a relatively simple graph. More importantly, *we repeatedly differentiate the same function computed in a node* for aggregation. For example, the differentiation of the variable $h(3, 1)$ is performed 16 times because it appears in 16 paths from x to o.

Obviously, this is an inefficient approach to compute gradients. For a network with 100 nodes in each layer and three layers, we will have a million paths. *Nevertheless, this is exactly what we do in traditional machine learning when our prediction function is a complex composition function.* Manually working out the details of a complex composition function is tedious and impractical beyond a certain level of complexity. It is here that one can apply dynamic programming (which is guided by the structure of the computational graph) in order to store important intermediate results. By using such an approach, one can minimize repeated computations, and achieve polynomial complexity.

2.3.2 Dynamic Programming for Computing Node-to-Node Derivatives

In graph theory, computing all types of path-aggregative values over directed acyclic graphs is done using dynamic programming. Consider a directed acyclic graph in which the value $z(i, j)$ (interpreted as local partial derivative of variable in node j with respect to variable in node i) is associated with edge (i, j). In other words, if $y(p)$ is the variable in the node p, we have the following:

$$z(i, j) = \frac{\partial y(j)}{\partial y(i)} \tag{2.7}$$

An example of such a computational graph is shown in Figure 2.6. In this case, we have associated the edge $(2, 4)$ with the corresponding partial derivative. We would like to compute the product of $z(i, j)$ over each path $P \in \mathcal{P}$ from source node s to output node t and then add them in order to obtain the partial derivative $S(s, t) = \frac{\partial y(t)}{\partial y(s)}$:

$$S(s, t) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j) \tag{2.8}$$

Let $A(i)$ be the set of nodes at the end points of outgoing edges from node i. We can compute the aggregated value $S(i, t)$ for each intermediate node i (between source node s and output node t) using the following well-known dynamic programming update:

$$S(i, t) \Leftarrow \sum_{j \in A(i)} S(j, t) z(i, j) \tag{2.9}$$

This computation can be performed backwards starting from the nodes directly incident on o, since $S(t, t) = \frac{\partial y(t)}{\partial y(t)}$ is already known to be 1. This is because the partial derivative of a variable with respect to itself is always 1. Therefore one can describe the pseudocode of this algorithm as follows:

Initialize $S(t, t) = 1$;
repeat
  Select an unprocessed node i such that the values of $S(j, t)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $S(i, t) \Leftarrow \sum_{j \in A(i)} S(j, t) z(i, j)$;
until all nodes have been selected;

Note that the above algorithm always selects a node i for which the value of $S(j, t)$ is available for all nodes $j \in A(i)$. Such a node is always available in directed acyclic graphs, and the node selection order will always be in the backwards direction starting from node

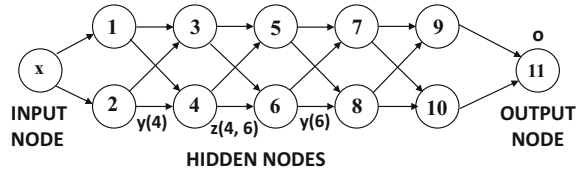


Figure 2.6: Example of computational graph with edges corresponding to local partial derivatives

t. Therefore, the above algorithm will work only when the computational graph does not have cycles.

The algorithm discussed above is used by the network optimization community for computing all types of path-centric functions between *source-sink* node pairs (s, t) on directed acyclic graphs, which would otherwise require exponential time. For example, one can even use a variation of the above algorithm to find the longest path in a directed acyclic graph [8]. When applied on computational graphs corresponding to neural networks, the above algorithm is referred to as the backpropagation algorithm.

Interestingly, *the aforementioned dynamic programming update is exactly the multivariate chain rule of Equation 2.3, which is repeated in the backwards direction starting at the output node where the local gradient is known.* This is because we derived the path-aggregative form of the loss gradient (Lemma 2.3.1) using this chain rule in the first place. The main difference is that we apply the rule in a particular order in order to minimize computations. Since this point is important, we summarize it below:

> Using dynamic programming to efficiently aggregate the product of local gradients along the exponentially many paths in a computational graph results in a dynamic programming update that is identical to the multivariate chain rule of differential calculus. The main point of dynamic programming is to apply this rule in a particular order, so that the derivative computations at different nodes are not repeated.

This approach is the backbone of the backpropagation algorithm used in neural networks. The above gradients are used to update the weights of the neural network. In the case where we have multiple output nodes $t_1, \ldots t_p$, one can initialize each $S(t_r, t_r)$ to 1, and then apply the same approach for each t_r.

Example of Computing Node-to-Node Derivatives

In order to show how the backpropagation approach works, we will provide an example of computation of node-to-node derivatives in a graph containing 10 nodes (see Figure 2.7). A variety of functions are computed in various nodes, such as the sum function (denoted by '+'), the product function (denoted by '*'), and the trigonometric sine/cosine functions. The variables in the 10 nodes are denoted by $y(1) \ldots y(10)$, where the variable $y(i)$ belongs to the ith node in the figure. Two of the edges incoming into node 6 also have the weights w_2 and w_3 associated with them. Other edges do not have weights associated with them.

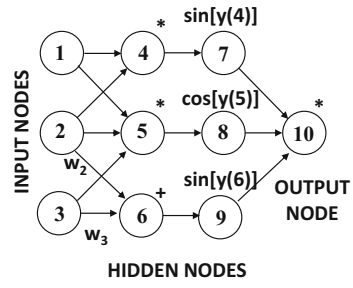


Figure 2.7: Example of node-to-node derivative computation

The functions computed in the various layers are as follows:

Layer 1: $y(4) = y(1) \cdot y(2)$, $y(5) = y(1) \cdot y(2) \cdot y(3)$, $y(6) = w_2 \cdot y(2) + w_3 \cdot y(3)$
Layer 2: $y(7) = \sin(y(4))$, $y(8) = \cos(y(5))$, $y(9) = \sin(y(6))$
Layer 3: $y(10) = y(7) \cdot y(8) \cdot y(9)$

We would like to compute the derivative of $y(10)$ with respect to each of the inputs $y(1)$, $y(2)$, and $y(3)$. One possibility is to simply express the $y(10)$ in closed form in terms of the inputs $y(1)$, $y(2)$ and $y(3)$, and then compute the derivative. By recursively using the above relationships, it is easy to show that $y(10)$ can be expressed in terms of $y(1)$, $y(2)$, and $y(3)$ as follows:

$$y(10) = \sin(y(1) \cdot y(2)) \cdot \cos(y(1) \cdot y(2) \cdot y(3)) \cdot \sin(w_2 \cdot y(2) + w_3 \cdot y(3))$$

As discussed earlier computing the closed-form derivative is not practical for larger networks. Furthermore, since one needs to compute the derivative of the output with respect to each and every node in the network, such an approach would also required closed-form expressions in terms of upstream nodes like $y(4)$, $y(5)$ and $y(6)$. All this tends to increase the amount of repeated computation. Luckily, backpropagation frees us from this repeated computation, since the derivative in $y(10)$ with respect to each and every node is computed by the backwards phase.

The algorithm starts, by initializing the derivative of the output $y(10)$ with respect to itself, which is 1:

$$S(10, 10) = \frac{\partial y(10)}{\partial y(10)} = 1$$

Subsequently, the derivatives of $y(10)$ with respect to all the variables on its incoming nodes are computed. Since $y(10)$ is expressed in terms of the variables $y(7)$, $y(8)$, and $y(9)$ incoming into it, this is easy to do, and the results are denoted by $z(7, 10)$, $z(8, 10)$, and $z(9, 10)$ (which is consistent with the notations used earlier in this chapter). Therefore, we have the following:

$$z(7, 10) = \frac{\partial y(10)}{\partial y(7)} = y(8) \cdot y(9)$$
$$z(8, 10) = \frac{\partial y(10)}{\partial y(8)} = y(7) \cdot y(9)$$
$$z(9, 10) = \frac{\partial y(10)}{\partial y(9)} = y(7) \cdot y(8)$$

Subsequently, we can use these values in order to compute $S(7, 10)$, $S(8, 10)$, and $S(9, 10)$ using the recursive backpropagation update:

$$S(7, 10) = \frac{\partial y(10)}{\partial y(7)} = S(10, 10) \cdot z(7, 10) = y(8) \cdot y(9)$$
$$S(8, 10) = \frac{\partial y(10)}{\partial y(8)} = S(10, 10) \cdot z(8, 10) = y(7) \cdot y(9)$$
$$S(9, 10) = \frac{\partial y(10)}{\partial y(9)} = S(10, 10) \cdot z(9, 10) = y(7) \cdot y(8)$$

Next, we compute the derivatives $z(4,7)$, $z(5,8)$, and $z(6,9)$ associated with all the edges incoming into nodes 7, 8, and 9:

$$z(4,7) = \frac{\partial y(7)}{\partial y(4)} = \cos[y(4)]$$
$$z(5,8) = \frac{\partial y(8)}{\partial y(5)} = -\sin[y(5)]$$
$$z(6,9) = \frac{\partial y(9)}{\partial y(6)} = \cos[y(6)]$$

These values can be used to compute $S(4,10)$, $S(5,10)$, and $S(6,10)$:

$$S(4,10) = \frac{\partial y(10)}{\partial y(4)} = S(7,10) \cdot z(4,7) = y(8) \cdot y(9) \cdot \cos[y(4)]$$
$$S(5,10) = \frac{\partial y(10)}{\partial y(5)} = S(8,10) \cdot z(5,8) = -y(7) \cdot y(9) \cdot \sin[y(5)]$$
$$S(6,10) = \frac{\partial y(10)}{\partial y(6)} = S(9,10) \cdot z(6,9) = y(7) \cdot y(8) \cdot \cos[y(6)]$$

In order to compute the derivatives with respect to the input values, one now needs to compute the values of $z(1,3)$, $z(1,4)$, $z(2,4)$, $z(2,5)$, $z(2,6)$, $z(3,5)$, and $z(3,6)$:

$$z(1,4) = \frac{\partial y(4)}{\partial y(1)} = y(2)$$
$$z(2,4) = \frac{\partial y(4)}{\partial y(2)} = y(1)$$
$$z(1,5) = \frac{\partial y(5)}{\partial y(1)} = y(2) \cdot y(3)$$
$$z(2,5) = \frac{\partial y(5)}{\partial y(2)} = y(1) \cdot y(3)$$
$$z(3,5) = \frac{\partial y(5)}{\partial y(3)} = y(1) \cdot y(2)$$
$$z(2,6) = \frac{\partial y(6)}{\partial y(2)} = w_2$$
$$z(3,6) = \frac{\partial y(6)}{\partial y(3)} = w_3$$

These partial derivatives can be backpropagated to compute $S(1,10)$, $S(2,10)$, and $S(3,10)$:

$$\begin{aligned} S(1,10) &= \frac{\partial y(10)}{\partial y(1)} = S(4,10) \cdot z(1,4) + S(5,10) \cdot z(1,5) \\ &= y(8) \cdot y(9) \cdot \cos[y(4)] \cdot y(2) - y(7) \cdot y(9) \cdot \sin[y(5)] \cdot y(2) \cdot y(3) \\ S(2,10) &= \frac{\partial y(10)}{\partial y(2)} = S(4,10) \cdot z(2,4) + S(5,10) \cdot z(2,5) + S(6,10) \cdot z(2,6) \\ &= y(8) \cdot y(9) \cdot \cos[y(4)] \cdot y(1) - y(7) \cdot y(9) \cdot \sin[y(5)] \cdot y(1) \cdot y(3) + \\ &\quad + y(7) \cdot y(8) \cdot \cos[y(6)] \cdot w_2 \end{aligned}$$

$$S(3,10) = \frac{\partial y(10)}{\partial y(3)} = S(5,10) \cdot z(3,5) + S(6,10) \cdot z(3,6)$$
$$= -y(7) \cdot y(9) \cdot \sin[y(5)] \cdot y(1) \cdot y(2) + y(7) \cdot y(8) \cdot \cos[y(6)] \cdot w_3$$

Note that the use of a backward phase has the advantage of computing the derivative of $y(10)$ (output node variable) with respect to all the hidden and input node variables. These different derivatives have many sub-expressions in common, although the derivative computation of these sub-expressions is not repeated. This is the advantage of using the backwards phase for derivative computation as opposed to the use of closed-form expressions.

Because of the tedious nature of the closed-form expressions for outputs, the algebraic expressions for derivatives are also very long and awkward (no matter how we compute them). One can see that this is true even for the simple computational graph of this section (containing just ten nodes). For example, if one examines the derivative of $y(10)$ with respect to each of nodes $y(1)$, $y(2)$ and $y(3)$, the algebraic expression wraps into multiple lines. Furthermore, one cannot avoid the presence of repeated subexpressions within the algebraic derivative. This is counter-productive because our original goal of the backwards algorithm was to avoid the repeated computation endemic to the use of traditional derivative evaluation with closed-form expressions. Therefore, one does not *algebraically* compute these types of expressions in real-world networks. One would first *numerically* compute all the node variables for a *specific* input. Subsequently, one would *numerically* carry the derivatives backward, so that one does not have to carry the large algebraic expressions (with many repeated sub-expressions) in the backwards direction. The advantage of carrying numerical expressions is that multiple terms get consolidated into a single numerical value, which is specific to a particular input. Although one must repeat the backwards computation algorithm for each training point, it is still a better choice than computing the (massive) symbolic derivative in one shot and substituting the values in different training points. This is the reason that such an approach is referred to as *numerical differentiation* rather than *symbolic differentiation*. In much of machine learning, one first computes the algebraic derivative before substituting numerical values of the variables in the expression (for the derivative) to perform gradient-descent updates. This is different from the case of computational graphs, where the backwards algorithm is *numerically* applied to each training point.

2.3.3 Converting Node-to-Node Derivatives into Loss-to-Weight Derivatives

Most computational graphs define loss functions with respect to output node variables. One needs to compute the derivatives with respect to weights on *edges* rather than the node variables (in order to update the weights). In general, the node-to-node derivatives can be converted into loss-to-weight derivatives with a few additional applications of the univariate and multivariate chain rule.

Consider the case in which we have computed the node-to-node derivative of output variables in nodes indexed by $t_1, t_2, \ldots t_p$ with respect to the variable in node i using the dynamic programming approach in the previous section. Therefore, the computational graph has p output nodes in which the corresponding variable values are $y(t_1) \ldots y(t_p)$ (since the indices of the output nodes are $t_1 \ldots t_p$). The loss function is denoted by $L(y(t_1), \ldots y(t_p))$. We would like to compute the derivative of this loss function with respect to *the weights in the incoming edges of* i. For the purpose of this discussion, let w_{ji} be the weight of an edge from node index j to node index i. Therefore, we want to compute the derivative of the loss

function with respect to w_{ji}. In the following, we will abbreviate $L(y_{t_1}, \ldots y_{t_p})$ with L for compactness of notation:

$$\frac{\partial L}{\partial w_{ji}} = \left[\frac{\partial L}{\partial y(i)}\right] \frac{\partial y(i)}{\partial w_{ji}} \qquad \text{[Univariate chain rule]}$$

$$= \left[\sum_{k=1}^{p} \frac{\partial L}{\partial y(t_k)} \frac{\partial y(t_k)}{\partial y(i)}\right] \frac{\partial y(i)}{\partial w_{ji}} \qquad \text{[Multivariate chain rule]}$$

Here, it is noteworthy that the loss function is typically a closed-form function of the variables in the node indices $t_1 \ldots t_p$, which (for example) could be a least-squares function. Therefore, each derivative of the loss L with respect to $y(t_i)$ is easy to compute. Furthermore, the value of each $\frac{\partial y(t_k)}{\partial y(i)}$ for $k \in \{1 \ldots p\}$ can be computed using the dynamic programming algorithm of the previous section. The value of $\frac{\partial y_i}{\partial w_{ji}}$ is a derivative of the **local** function at each node, which usually has a simple form. Therefore, the loss-to-weight derivatives can be computed relatively easily, once the node-to-node derivatives have been computed using dynamic programming.

Although one can apply the pseudocode of page 38 to compute $\frac{\partial y(t_k)}{\partial y(i)}$ for each $k \in \{1 \ldots p\}$, it is more efficient to collapse all these computations into a single backwards algorithm. In practice, one initializes the derivatives at the output nodes to the loss derivatives $\frac{\partial L}{\partial y(t_k)}$ for each $k \in \{1 \ldots p\}$ rather than the value of 1 (as shown in the pseudocode of page 38). Subsequently, the entire loss derivative $\Delta(i) = \frac{\partial L}{\partial y(i)}$ is propagated backwards. Therefore, the modified algorithm for computing the loss derivative with respect to the *node variables* as well as the *edge variables* is as follows:

Initialize $\Delta(t_r) = \frac{\partial L}{\partial y(t_k)}$ for each $k \in \{1 \ldots p\}$;
repeat
  Select an unprocessed node i such that the values of $\Delta(j)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $\Delta(i) \Leftarrow \sum_{j \in A(i)} \Delta(j) z(i, j)$;
until all nodes have been selected;
for each edge (j, i) with weight w_{ji} **do** compute $\frac{\partial L}{\partial w_{ji}} = \Delta(i) \frac{\partial y(i)}{\partial w_{ji}}$

In the above algorithm, $y(i)$ denotes the variable at node i. The key difference from the algorithm on page 38 is in the nature of the initialization and the addition of a final step computing the edge-wise derivatives. However, the core algorithm for computing the node-to-node derivatives remains an integral part of this algorithm. In fact, one can convert all the weights on the edges into additional "input" nodes containing weight parameters, and also add computational nodes that multiply the weights with the corresponding variables at the tail nodes of the edges. Furthermore, a computational node can be added that computes the loss from the output node(s). For example, the architecture of Figure 2.7 can be converted to that in Figure 2.8. Performing only node-to-node derivative computation in Figure 2.8 from the loss node to the weight nodes is equivalent to loss-to-weight derivative computation. In other words, *loss-to-weight derivative computation in a weighted graph is equivalent to node-to-node derivative computation in a modified computational graph.* The derivative of the loss with respect to each weight can be denoted by the vector $\frac{\partial L}{\partial \overline{W}}$ (in matrix calculus[3]

[3]In matrix calculus, the gradient of a scalar with respect to a vector is a vector of the same length in which the ith component is the partial derivative of the scalar with respect to the ith component of the vector. With appropriate choice of convention, the gradient of scalar with respect to a column vector maps to a column vector.

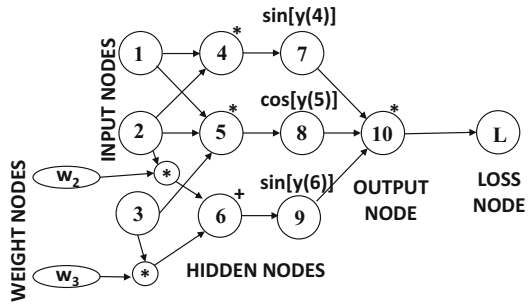


Figure 2.8: Converting loss-to-weight derivatives into node-to-node derivative computation based on Figure 2.7. Note the extra weight nodes and an extra loss node where loss is computed.

notation), where $\overline{W}$ denotes the weight vector. Subsequently, the standard gradient descent update can be performed:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}} \tag{2.10}$$

Here, α is the *learning rate* that controls the size of the steps. This type of update is performed to convergence by repeating the process with different inputs in order to learn the weights of the computational graph.

Example of Computing Loss-to-Weight Derivatives

Consider the case of Figure 2.7 in which the loss function is defined by $L = \log[y(10)^2]$, and we wish to compute the derivative of the loss with respect to the weights w_2 and w_3. In such a case, the derivative of the loss with respect to the weights is given by the following:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y(10)} \frac{\partial y(10)}{\partial y(6)} \frac{\partial y(6)}{\partial w_2} = \left[\frac{2}{y(10)}\right] [y(7) \cdot y(8) \cdot \cos[y(6)]]\, y(2)$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial y(10)} \frac{\partial y(10)}{\partial y(6)} \frac{\partial y(6)}{\partial w_3} = \left[\frac{2}{y(10)}\right] [y(7) \cdot y(8) \cdot \cos[y(6)]]\, y(3)$$

Note that the quantity $\frac{\partial y(10)}{\partial y(6)}$ has been obtained using the example in the previous section on node-to-node derivatives. In practice, these quantities are not computed algebraically. This is because the aforementioned algebraic expressions can be extremely awkward for large networks. Rather, for each numerical input set $\{y(1), y(2), y(3)\}$, one computes the different values of $y(i)$ in a forward phase. Subsequently, the derivatives of the loss with respect to each node variable (and incoming weights) are computed in a backwards phase. Again, these values are computed numerically for a specific input set $\{y(1), y(2), y(3)\}$. The numerical gradients can be used in order to update the weights for learning purposes.

2.4 Backpropagation in Neural Networks

In this section, we will describe how the generic algorithm based on computational graphs can be used in order to perform the backpropagation algorithm in neural networks. The key idea is that specific variables in the neural network need to be defined as nodes of the

computational-graph abstraction. The same neural network can be represented by different types of computational graphs, depending on which variables in the neural network are used to create computational graph nodes. The precise methodology for performing the backpropagation updates depends heavily on this design choice.

Consider the case of a neural network that first applies a linear function with weights w_{ij} on its inputs to create the pre-activation value $a(i)$, and then applies the activation function $\Phi(\cdot)$ in order to create the output $h(i)$:

$$h(i) = \Phi(a(i))$$

The variables $h(i)$ and $a(i)$ are shown in Figure 2.9. In this case, it is noteworthy that there are several ways in which the computational graph can be created. For example, one might create a computational graph in which each node contains the post-activation value $h(i)$, and therefore we are implicitly setting $y(i) = h(i)$. A second choice is to create a computational graph in which each node contains the pre-activation variable $a(i)$ and therefore we are setting $y(i) = a(i)$. It is even possible to create a decoupled computational graph containing both $a(i)$ and $h(i)$; in the last case, the computational graph will have twice as many nodes as the neural network. In all these cases, a relatively straightforward special-case/simplification of the pseudocodes in the previous section can be used for learning the gradient:

1. The post-activation value $y(i) = h(i)$ could represent the variable in the ith computational node in the graph. Therefore, each computational node in such a graph *first* applies the linear function, *and then* applies the activation function. The post-activation value is shown in Figure 2.9. In such a case, the value of $z(i, j) = \frac{\partial y(j)}{\partial y(i)} = \frac{\partial h(j)}{\partial h(i)}$ in the pseudocode of page 43 is $w_{ij}\Phi'_j$. Here, w_{ij} is the weight of the edge from i to j and $\Phi'_j = \frac{\partial \Phi(a(j))}{\partial a(j)}$ is the local derivative of the activation function at node j with respect to its argument. The value of each $\Delta(t_r)$ at output node t_r is simply the derivative of the loss function with respect to $h(t_r)$. The final derivative with respect to the weight w_{ji} (in the final line of the pseudocode on page 43) is equal to $\Delta(i)\frac{\partial h(i)}{\partial w_{ji}} = \Delta(i)h(j)\Phi'_i$.

2. The pre-activation value (after applying the linear function), which is denoted by $a(i)$, could represent the variable in each computational node i in the graph. Note the subtle distinction between the work performed in computational nodes and neural network nodes. Each *computational* node *first* applies the activation function to each of its inputs before applying a linear function, whereas these operations are performed in the reverse order in a neural network. The structure of the computational graph is roughly similar to the neural network, except that the first layer of computational

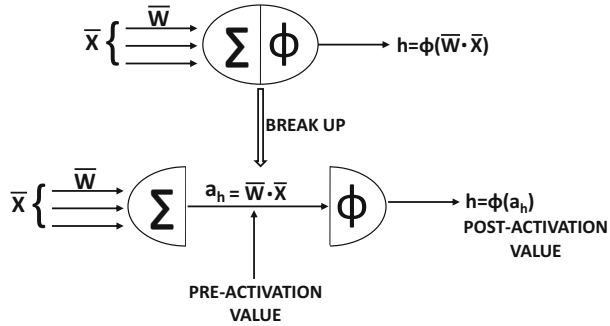


Figure 2.9: Pre- and post-activation values within a neuron

nodes do not contain an activation. In such a case, the value of $z(i, j) = \frac{\partial y(j)}{\partial y(i)} = \frac{\partial a(j)}{\partial a(i)}$ in the pseudocode of page 43 is $\Phi_i' w_{ij}$. Note that $\Phi(a(i))$ is being differentiated with respect to its argument in this case, rather than $\Phi(a(j))$ as in the case of the post-activation variables. The value of the loss derivative with respect to the pre-activation variable $a(t_r)$ in the rth output node t_r needs to account for the fact that it is a pre-activation value, and therefore, we cannot directly use the loss derivative with respect to post-activation values. Rather the post-activation loss derivative needs to be *multiplied with the derivative* Φ_{t_r}' *of the activation function at that node*. The final derivative with respect to the weight w_{ji} (final line of pseudocode on page 43) is equal to $\Delta(i)\frac{\partial a(i)}{\partial w_{ji}} = \Delta(i)h(j)$.

The use of pre-activation variables for backpropagation is more common than the use of post-activation variables. Therefore, we present the backpropagation algorithm in a crisp pseudocode with the use of pre-activation variables. Let t_r be the index of the rth output node. Then, the backpropagation algorithm with pre-activation variables may be presented as follows:

Initialize $\Delta(t_r) = \frac{\partial L}{\partial y(t_r)} = \Phi'(a(t_r))\frac{\partial L}{\partial h(t_r)}$ for each output node t_r with $r \in \{1 \ldots k\}$;
repeat
  Select an unprocessed node i such that the values of $\Delta(j)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $\Delta(i) \Leftarrow \Phi_i' \sum_{j \in A(i)} w_{ij}\Delta(j)$;
until all nodes have been selected;
for each edge (j, i) with weight w_{ji} **do** compute $\frac{\partial L}{\partial w_{ji}} = \Delta(i)h(j)$;

It is also possible to use both pre-activation and post-activation variables as separate nodes of the computational graph. In section 2.5.3, we will combine this approach with a vector-centric representation.

The backpropagation algorithm is quite general in terms of the types of architectures where it can be used. It is most commonly used in feed-forward neural networks, where all input nodes feed into the first hidden layer, the hidden layers successively feed into one another, and the final hidden layer feeds into the output layer. The hidden layer generally does not take inputs, and the loss is generally not computed over the values in the hidden layers. Because of this focus, it is easy to forget that the backpropagation algorithm can be used on *any type of parameterized computational graph*, as long as it is acyclic. For example, a neural network is proposed in [535] that allows random bags of features as direct inputs to hidden layers (see Figure 2.10). Even in this case, the backpropagation pseudocode discussed above would work because it only requires a directed acyclic graph. Furthermore, one could easily experiment with any type of function being used in a computational node (beyond well-known linear summation and activation functions), as long as it can be differentiated.

2.4.1 Some Useful Derivatives of Activation Functions

All of the updates in the previous section require the use of the derivatives of various activation functions. For this reason, the derivatives of these activation functions are used repeatedly in this book. This section provides details of these derivatives.

1. *Linear and sign activations:* The derivative of the linear activation function is 1 at all places. The derivative of sign(v) is 0 at all values of v other than at $v = 0$, where it is discontinuous and non-differentiable. Because of the zero gradient and non-differentiability of this activation function, it is rarely used in the loss function

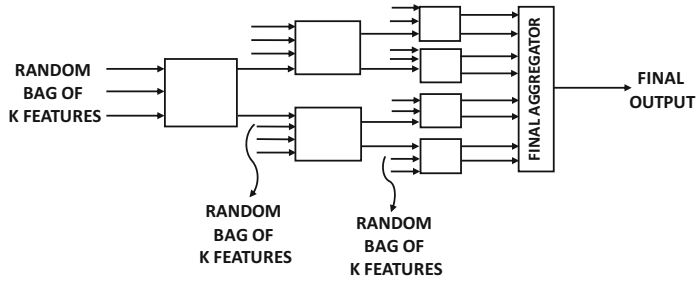
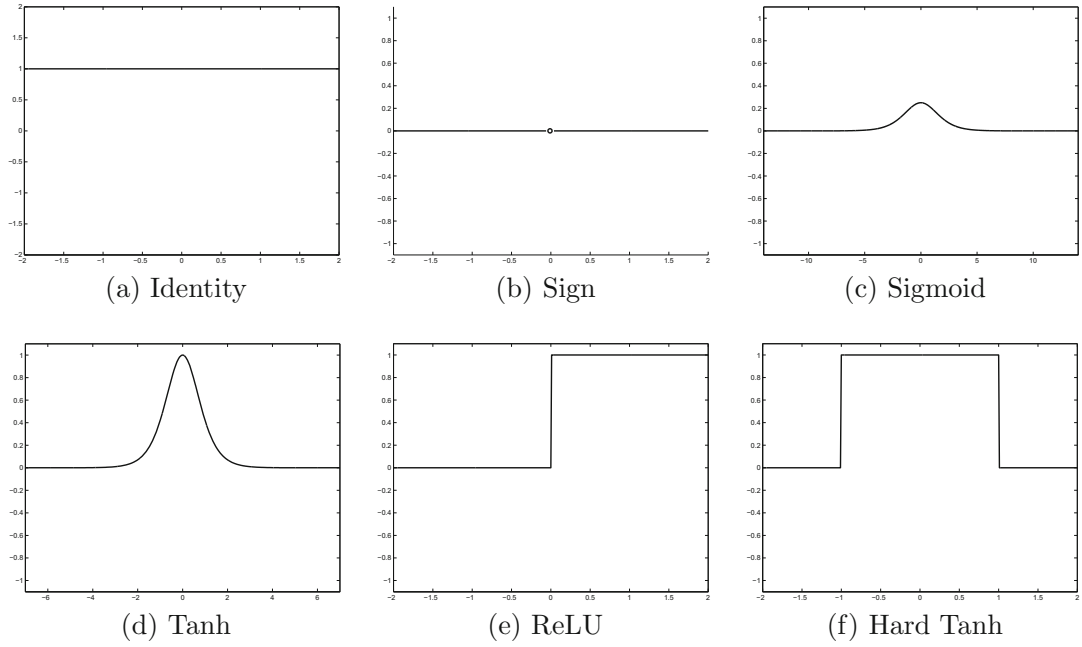


Figure 2.10: An example of an unconventional architecture in which inputs occur to layers other than the first hidden layer.



(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Figure 2.11: The derivatives of various activation functions

even when it is used for prediction at testing time. The derivatives of the linear and sign activations are illustrated in Figures 2.11(a) and (b), respectively.

2. *Sigmoid activation:* The derivative of sigmoid activation is particularly simple, when it is expressed in terms of the *output* of the sigmoid, rather than the input. Let o be the output of the sigmoid function with argument v:

$$o = \frac{1}{1 + \exp(-v)} \tag{2.11}$$

Then, one can write the derivative of the activation as follows:

$$\frac{\partial o}{\partial v} = \frac{\exp(-v)}{(1 + \exp(-v))^2} \tag{2.12}$$

The key point is that this sigmoid can be written more conveniently in terms of the outputs:

$$\frac{\partial o}{\partial v} = o(1 - o) \tag{2.13}$$

The derivative of the sigmoid is often used as a function of the output rather than the input. The derivative of the sigmoid activation function is illustrated in Figure 2.11(c).

3. *Tanh activation:* As in the case of the sigmoid activation, the tanh activation is often used as a function of the output o rather than the input v:

$$o = \frac{\exp(2v) - 1}{\exp(2v) + 1} \tag{2.14}$$

One can then compute the derivative as follows:

$$\frac{\partial o}{\partial v} = \frac{4 \cdot \exp(2v)}{(\exp(2v) + 1)^2} \tag{2.15}$$

One can also write this derivative in terms of the output o:

$$\frac{\partial o}{\partial v} = 1 - o^2 \tag{2.16}$$

The derivative of the tanh activation is illustrated in Figure 2.11(d).

4. *ReLU and hard tanh activations:* The ReLU takes on a partial derivative value of 1 for non-negative values of its argument, and 0, otherwise. The hard tanh function takes on a partial derivative value of 1 for values of the argument in $[-1, +1]$ and 0, otherwise. The derivatives of the ReLU and hard tanh activations are illustrated in Figures 2.11(e) and (f), respectively.

2.4.2 Examples of Updates for Various Activations

Since updates with pre-activation variables are more common, we provide the concrete updates with the use of pre-activation variables. We repeat the update discussed in the pseudocode of the previous section with pre-activation variables:

$$\Delta(i) \Leftarrow \Phi_i' \sum_{j \in A(i)} w_{ij} \Delta(j)$$

Since $\Delta(i)$ corresponds to the pre-activation variables, the post-activation values are denoted by $h(i) = \Phi(a(i))$. In the following, we provide the instantiation of the above pre-activation update with the use of different types of activation functions. Note that these updates are always expressed in terms of post-activation values $h(i) = \Phi[a(i)]$, even though the computational graph implicitly uses pre-activation variables:

$$\Delta(i) \Leftarrow \sum_{j \in A(i)} w_{ij} \Delta(j) \quad \text{[Linear]}$$

$$\Delta(i) \Leftarrow h(i)[1 - h(i)] \sum_{j \in A(i)} w_{ij} \Delta(j) \quad \text{[Sigmoid]}$$

$$\Delta(i) \Leftarrow [1 - h(i)^2] \sum_{j \in A(i)} w_{ij} \Delta(j) \quad [\text{Tanh}]$$

Note that the derivative of the sigmoid can be written in terms of its *output* value $h(i)$ as $h(i)[1 - h(i)]$. Similarly, the tanh derivative can be expressed as $[1 - h(i)^2]$. The derivatives of different activation functions are discussed in section 2.4.1 of Chapter 1. For the ReLU function, the value of $\Delta(i)$ can be computed in case-wise fashion:

$$\Delta(i) \Leftarrow \begin{cases} \sum_{j \in A(i)} w_{ij} \Delta(j) & \text{if } 0 < a(i) \\ 0 & \text{otherwise} \end{cases}$$

A similar recurrence can be shown for the hard tanh function except that the update condition is slightly different:

$$\Delta(i) \Leftarrow \begin{cases} \sum_{j \in A(i)} w_{ij} \Delta(j) & \text{if } -1 < a(i) < 1 \\ 0 & \text{otherwise} \end{cases}$$

It is noteworthy that the ReLU and tanh are non-differentiable at exactly the condition boundaries. However, this is rarely a problem in practical settings, in which one works with finite precision.

The Special Case of Softmax

Softmax activation is a special case because the function is not computed with respect to one input, but with respect to multiple inputs. Therefore, one cannot use exactly the same type of update, as with other activation functions. As discussed in Equation 1.7 of Chapter 1, the softmax activation function is a vector-to-vector function that converts k real-valued predictions $v_1 \ldots v_k$ into output probabilities $o_1 \ldots o_k$ using the following relationship:

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\} \tag{2.17}$$

Note that if we try to use the chain rule to backpropagate the derivative of the loss L with respect to $v_1 \ldots v_k$, then one has to compute each $\frac{\partial L}{\partial o_i}$ and also each $\frac{\partial o_i}{\partial v_j}$. This backpropagation of the softmax is greatly simplified, when we take two facts into account:

1. The softmax is almost always used in the output layer.
2. The softmax is almost always paired with the *cross-entropy loss*. If $y_1 \ldots y_k \in \{0, 1\}$, be the one-hot encoded (observed) outputs for the k mutually exclusive classes, then the cross-entropy loss is defined as follows:

$$L = -\sum_{i=1}^{k} y_i \log(o_i) \tag{2.18}$$

The key point is that the value of $\frac{\partial L}{\partial v_i}$ has a particularly simple form in the case of the softmax:

$$\frac{\partial L}{\partial v_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial o_j} \cdot \frac{\partial o_j}{\partial v_i} = o_i - y_i \tag{2.19}$$

The reader is encouraged to work out the detailed derivation of the result above; it is tedious, but relatively straightforward algebra. The derivation is enabled by the fact that the value of $\frac{\partial o_j}{\partial v_i}$ in Equation 2.19 can be shown to be equal to $o_i(1 - o_i)$ when $i = j$ (which is the same as sigmoid), and otherwise can be shown to be equal to $-o_i o_j$ (see Exercise 7).

In this case, we have decoupled the backpropagation update of the softmax activation from the updates of the weighted layers. In general, it is helpful to create a view of backpropagation in which the linear matrix multiplications and activation layers are decoupled because it greatly simplifies the updates. This view will be discussed in the next section.

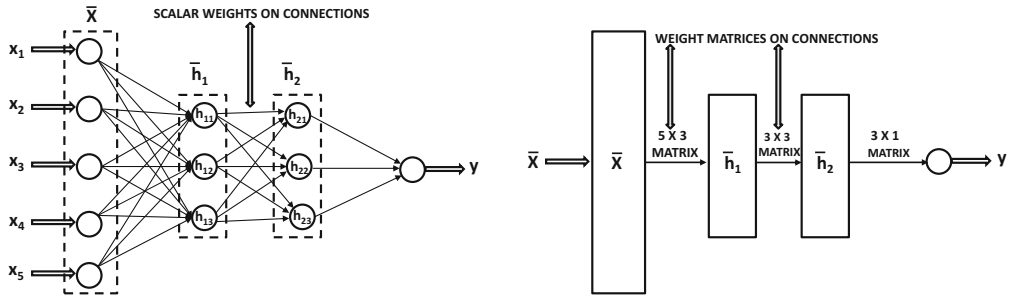
2.5 The Vector-Centric View of Backpropagation

In this section, we will discuss the vector-centric view of backpropagation. First, we recap the vector architecture of neural networks discussed in Chapter 1. The corresponding $(k + 1)$ weight matrices between successive layers are denoted by $W_1 \ldots W_{k+1}$. Let $\overline{x}$ be the d-dimensional column vector corresponding to the input, $\overline{h}_1 \ldots \overline{h}_k$ be the column vectors corresponding to the hidden layers, and $\overline{o}$ be the m-dimensional column vector corresponding to the output. Then, we have the following recurrence condition for multi-layer networks:

$$\overline{h}_1 = \Phi(W_1 \overline{x}) = W_1 \overline{x}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1} \overline{h}_p) = W_{p+1} \overline{h}_p \quad \forall p \in \{1 \ldots k-1\}$$
$$\overline{o} = \Phi(W_{k+1} \overline{h}_k) = W_{k+1} \overline{h}_k$$

The function $\Phi(\cdot)$ is applied in element-wise fashion. A key problem here is that each of the above functions is a *vector composition* of a linear layer and a nonlinear activation layer. Furthermore, the output is an even more complex composition function of earlier layers.

A comparison of the scalar architecture is provided with the vector architecture in Figure 2.12. It is noteworthy that the *connection matrix* between the input layer and the first hidden layer is of size 5×3, since there are 5 inputs. However, in order to apply the linear transformation $W_1 \overline{x}$ to the 5-dimensional column vector $\overline{x}$, the weight matrix will be of size 3×5, so that $\Phi(W_1 \overline{x})$ is a 3-dimensional vector. A key point is that the entire neural network can be expressed as a single path with vector-centric operations, which greatly simplifies the topology of the computational graph. However, all functions in the vector-centric view of neural networks are vector-to-vector functions. Therefore, one needs to use the vector-to-vector derivatives and a corresponding chain rule. With this machinery,



(a) Scalar notation and architecture (b) Vector notation and architecture

Figure 2.12: A feed-forward network with two hidden layers and a single output layer.

backpropagation becomes much simpler, as one has an extremely simple topology of a single path (at least in the case of layer-wise neural networks).

2.5.1 Derivatives with Respect to Vectors

The backpropagation algorithm requires the computation of node-to-node derivatives and loss-to-node derivatives as intermediate steps. In the vector-centric view, one wants to compute derivatives with respect to entire layers of nodes. To do so, one must use *matrix calculus* notation, which allows derivatives of scalars and vectors with respect to other vectors.

For example, consider the case where one wishes to compute the derivative of a scalar loss L with respect to a vector layer $\overline{x}$, where $\overline{x} = [x_1 \ldots x_d]^T$ is a d-dimensional column vector. The derivative of a scalar with respect to a column vector is another column vector, using the *denominator layout convention*[4] of matrix calculus. This derivative is denoted by $\frac{\partial L}{\partial \overline{x}}$, and is simply the gradient. This notation is a scalar-to-vector derivative, which always returns a vector. Therefore, we have the following:

$$\nabla L = \frac{\partial L}{\partial \overline{x}} = \left[\frac{\partial L}{\partial x_1} \cdots \frac{\partial L}{\partial x_d}\right]^T$$

The matrix calculus notation also allows derivatives of vectors with respect to vectors. For example, the derivative of an m-dimensional column vector $\overline{h} = [h_1, \ldots, h_m]^T$ with respect to a d-dimensional column vector $\overline{x} = [x_1, \ldots, x_d]^T$ is a $d \times m$ matrix in the denominator layout convention. The (i, j)th entry of this matrix is the derivative of h_j with respect to x_i:

$$\left[\frac{\partial \overline{h}}{\partial \overline{x}}\right]_{ij} = \frac{\partial h_j}{\partial x_i} \tag{2.20}$$

This matrix is closely related to the *Jacobian matrix* in matrix calculus. The (i, j)th element of the Jacobian is always $\frac{\partial h_i}{\partial w_j}$, and therefore it is the transpose of the matrix $\frac{\partial \overline{h}}{\partial \overline{x}}$ shown in Equation 2.20:

$$\text{Jacobian}(\overline{h}, \overline{x}) = \left[\frac{\partial \overline{h}}{\partial \overline{x}}\right]^T$$

The transposition occurs because of the use of denominator layout. In fact the Jacobian matrix is exactly equal to $\frac{\partial \overline{h}}{\partial \overline{x}}$ in the numerator layout convention of matrix calculus. However, we will consistently work with the denominator convention in this book.

Two special cases of vector-to-vector derivatives are very useful in neural networks. The first is the linear propagation $\overline{h} = W\overline{x}$, which occurs in linear layers of neural networks. In such a case, $\frac{\partial \overline{h}}{\partial \overline{x}}$ can be shown to be W^T. The second is when an *element-wise* activation function $\overline{h} = \Phi(\overline{x})$ is applied to the d-dimensional input vector $\overline{x}$ to create another d-dimensional vector. In such a case, $\frac{\partial \overline{h}}{\partial \overline{x}}$ can be shown to be the $d \times d$ diagonal matrix in which the ith diagonal entry is the derivative $\Phi'(x_i) = \frac{\partial \Phi(x_i)}{\partial x_i}$. Here, x_i is the ith component of the vector $\overline{x}$.

2.5.2 Vector-Centric Chain Rule

As discussed above, a layered neural network uses repeated compositions of vector functions. Therefore, the function computed by a layered neural network is of the nested form

[4] In the numerator layout convention, the derivative of a scalar with respect to a column vector is a row vector. The choice of convention provides equivalent results with some transpositional changes.

$F_k(F_{k-1}(\ldots F_1(\overline{x})))$. The vectored chain rule is useful for computing derivatives of such functions, and is defined as follows:

Theorem 2.5.1 (Vectored Chain Rule) *Consider a composition function of the following form:*

$$\overline{o} = F_k(F_{k-1}(\ldots F_1(\overline{x})))$$

Assume that each $F_i(\cdot)$ takes as input an n_i-dimensional column vector and outputs an n_{i+1}-dimensional column vector. Therefore, the input $\overline{x}$ is an n_1-dimensional vector and the final output $\overline{o}$ is an n_{k+1}-dimensional vector. For brevity, denote the vector output of $F_i(\cdot)$ by $\overline{h}_i$. Then, the vectored chain rule asserts the following:

$$\underbrace{\left[\frac{\partial \overline{o}}{\partial \overline{x}}\right]}_{n_1 \times n_{k+1}} = \underbrace{\left[\frac{\partial \overline{h}_1}{\partial \overline{x}}\right]}_{n_1 \times n_2} \underbrace{\left[\frac{\partial \overline{h}_2}{\partial \overline{h}_1}\right]}_{n_2 \times n_3} \cdots \underbrace{\left[\frac{\partial \overline{h}_{k-1}}{\partial \overline{h}_{k-2}}\right]}_{n_{k-1} \times n_k} \underbrace{\left[\frac{\partial \overline{o}}{\partial \overline{h}_{k-1}}\right]}_{n_k \times n_{k+1}}$$

It is easy to see that the size constraints of matrix multiplication are respected in this case.

2.5.3 A Decoupled View of Vector-Centric Backpropagation

In the previous discussion, two equivalent ways of computing the updates based on pre-activation and post-activation values are provided. In each case, *one is really backpropagating through a linear matrix multiplication and an activation computation simultaneously.* However, in many real implementations, the linear computations and the activation computations are decoupled as separate "layers," and one separately backpropagates through the two layers. Furthermore, we use a vector-centric representation of the neural network, so that operations on vector representations of layers are vector-to-vector operations such as a matrix multiplication in a linear layer (cf. Figure 2.12). This view greatly simplifies the computations, since the entire neural network is a single path (albeit a vector-centric one). Therefore, one can create a neural network in which activation layers are alternately arranged with linear layers, as shown in Figure 2.13. Note that the activation layers can use identity activation if needed. Activation layers (usually) perform one-to-one, element-wise computations on the vector components with the activation function $\Phi(\cdot)$, whereas linear layers perform all-to-all computations by multiplying with the coefficient matrix W. Then, for each pair of matrix multiplication and activation function layers, the following forward and backward steps need to be performed:

1. Let $\overline{z}_i$ and $\overline{z}_{i+1}$ be the column vectors of activations in the forward direction when the matrix of linear transformations from the ith to the $(i+1)$th layer is denoted[5] by W. Furthermore, let $\overline{g}_i$ and $\overline{g}_{i+1}$ be the backpropagated vectors of gradients in the two layers. Each element of $\overline{g}_i$ is the partial derivative of the loss function with respect to a hidden variable in the ith layer. Then, we have the following:

$$\overline{z}_{i+1} = W\overline{z}_i \qquad \text{[Forward Propagation]}$$
$$\overline{g}_i = W^T \overline{g}_{i+1} \qquad \text{[Backward Propagation]}$$

[5]Strictly speaking, we should use the notation W_i instead of W, although we omit the subscripts here for simplicity, since the entire discussion in this section is devoted to the linear transforms between a single pair of layers.

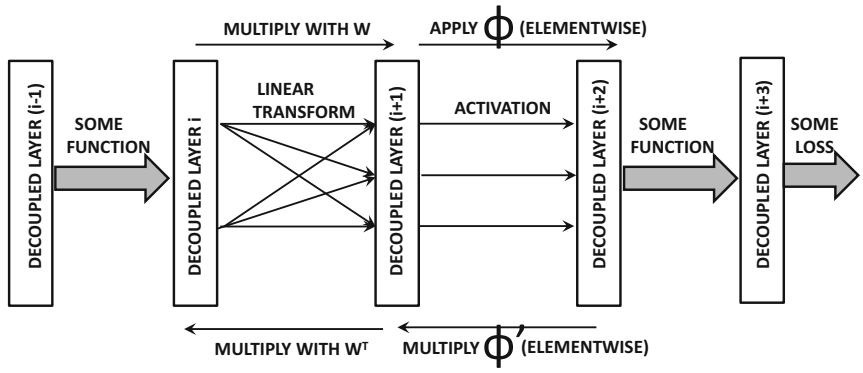


Figure 2.13: A decoupled view of backpropagation

Table 2.1: Examples of different functions and their backpropagation updates between layers i and $(i+1)$. The hidden values and gradients in layer i are denoted by $\overline{z}_i$ and $\overline{g}_i$. Some of these computations use $I(\cdot)$ as the binary indicator function.

Function	Type	Forward	Backward
Linear	Many-Many	$\overline{z}_{i+1} = W\overline{z}_i$	$\overline{g}_i = W^T\overline{g}_{i+1}$
Sigmoid	One-One	$\overline{z}_{i+1}$ =sigmoid$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (\overline{1} - \overline{z}_{i+1})$
Tanh	One-One	$\overline{z}_{i+1}$ =tanh$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (\overline{1} - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	One-One	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	One-One	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Many-One	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	Anything	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T\overline{g}_{i+1}$ J is Jacobian (Equation 2.21)

2. Now consider a situation where the activation function $\Phi(\cdot)$ is applied to each node in layer $(i+1)$ to obtain the activations in layer $(i+2)$. Then, we have the following:

$$\overline{z}_{i+2} = \Phi(\overline{z}_{i+1}) \quad \text{[Forward Propagation]}$$
$$\overline{g}_{i+1} = \overline{g}_{i+2} \odot \Phi'(\overline{z}_{i+1}) \quad \text{[Backward Propagation]}$$

Here, $\Phi_i(\cdot)$ and its derivative $\Phi_i'(\cdot)$ are applied in element-wise fashion to vector arguments. The symbol $\odot$ indicates elementwise multiplication.

Note the extraordinary simplicity once the activation is decoupled from the matrix multiplication in a layer. The forward and backward computations are shown in Figure 2.13. Furthermore, the derivatives of $\Phi(\overline{z}_{i+1})$ can often be computed in terms of the outputs of the next layer. Based on section 2.4.2, one can show the following for sigmoid activations:

$$\Phi'(\overline{z}_{i+1}) = \Phi(\overline{z}_{i+1}) \odot (1 - \Phi(\overline{z}_{i+1}))$$
$$= \overline{z}_{i+2} \odot (1 - \overline{z}_{i+2})$$

Examples of different types of backpropagation updates for various forward functions are shown in Table 2.1. In this case, we have used layer indices of i and $(i+1)$ for *both* linear

transformations and activation functions (rather than using $(i+2)$ for activation function). Note that the second to last entry in the table corresponds to the maximization function. This type of function is useful for *max-pooling* operations in convolutional neural networks.

Given the vector of gradients in a layer, one only has to apply the operations shown in the final column of Table 2.1 to obtain the gradients with respect to the previous layer. In the table, the vector indicator function $I(\overline{x} > 0)$ is an *element-wise* indicator function that returns a binary vector of the same size as $\overline{x}$; the ith output component is set to 1 when the ith component of $\overline{x}$ is larger than 0. The notation $\overline{1}$ indicates a column vector of 1s.

An arbitrary function on the ith layer of the network (beyond the simple linear matrix multiplications and element-wise activations) can be handled by assuming that the kth activation in layer-$(i+1)$ is obtained by applying an arbitrary function $f_k(\overline{z}_i)$ on the vector $\overline{z}_i$ of activations in layer-i. Then, backpropagation can be performed with the help of the Jacobian matrix $J = [J_{kr}]$, which is defined as follows:

$$J_{kr} = \frac{\partial f_k(\overline{z}_i)}{\partial \overline{z}_i^{(r)}} \tag{2.21}$$

Here, $\overline{z}_i^{(r)}$ is the rth element in $\overline{z}_i$. Let J be the matrix whose elements are J_{kr}. Then, it is easy to see that the backpropagation update from layer-$(i+1)$ to layer-i can be written as follows:

$$\overline{g}_i = J^T \overline{g}_{i+1} \tag{2.22}$$

Writing backpropagation equations as matrix multiplications is often beneficial from an implementation-centric point of view, such as acceleration with Graphics Processor Units (cf. section 4.9.1).

Derivatives with Respect to Weight Matrices

Upon performing the backpropagation, one only obtains the loss-to-node derivatives but not the loss-to-weight derivatives. Note that the elements in $\overline{g}_i$ represent gradients of the loss with respect to the *activations* in the ith layer, and therefore an additional step is needed to compute gradients with respect to the *weights*. The gradient of the loss with respect to a weight between the pth unit of the $(i-1)$th layer and the qth unit of ith layer is obtained by multiplying the pth element of $\overline{z}_{i-1}$ with the qth element of $\overline{g}_i$. One can also achieve this goal using a vector-centric approach, by simply computing the *outer product* of $\overline{g}_i$ and $\overline{z}_{i-1}$. In other words, the entire matrix M of derivatives of the loss with respect to the elements of the weight matrix between the $(i-1)$th layer and the ith layer is given by the following:

$$M = \overline{g}_i \overline{z}_{i-1}^T \tag{2.23}$$

Since M is given by the product of a column vector and a row vector of sizes equal to two successive layers, it is a matrix of exactly the same size as the weight matrix between the two layers. The (q, p)th element of M yields the derivative of the loss with respect to the weight between the pth element of $\overline{z}_{i-1}$ and qth element of $\overline{z}_i$.

Example of Vector-Centric Backpropagation

In order to explain vector-specific backpropagation, we will use an example in which the linear layers and activation layers have been decoupled. Figure 2.14 shows an example of

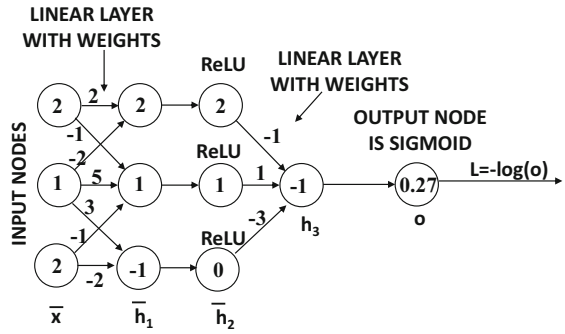


Figure 2.14: Example of decoupled neural network with vector layers $\overline{x}$, $\overline{h}_1$, $\overline{h}_2$, h_3, and o: variable values are shown within the nodes

a neural network with two computational layers, but they appear as four layers, since the activation layers have been decoupled as separated layers from the linear layers. The vector for the input layer is denoted by the 3-dimensional column vector $\overline{x}$, and the vectors for the computational layers are $\overline{h}_1$ (3-dimensional), $\overline{h}_2$ (3-dimensional), h_3 (1-dimensional), and output layer o (1-dimensional). The loss function is $L = -\log(o)$. These notations are annotated in Figure 2.14. The input vector $\overline{x}$ is $[2, 1, 2]^T$, and the weights of the edges in the two linear layers are annotated in Figure 2.14. Missing edges between $\overline{x}$ and $\overline{h}_1$ are assumed to have zero weight. In the following, we will provide the details of both the forward and the backwards phase.

Forward phase: The first hidden layer $\overline{h}_1$ is related to the input vector $\overline{x}$ with the weight matrix W as $\overline{h}_1 = W\overline{x}$. We can reconstruct the weights matrix W and then compute $\overline{h}_1$ for forward propagation as follows:

$$W = \begin{bmatrix} 2 & -2 & 0 \\ -1 & 5 & -1 \\ 0 & 3 & -2 \end{bmatrix}; \quad \overline{h}_1 = W\overline{x} = \begin{bmatrix} 2 & -2 & 0 \\ -1 & 5 & -1 \\ 0 & 3 & -2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

The hidden layer $\overline{h}_2$ is obtained by applying the ReLU function in element-wise fashion to $\overline{h}_1$ during the forward phase. Therefore, we obtain the following:

$$\overline{h}_2 = \text{ReLU}(\overline{h}_1) = \text{ReLU} \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$$

Subsequently, the 1×3 weight matrix $W_2 = [-1, 1, -3]$ is used to transform the 3-dimensional vector $\overline{h}_2$ to the 1-dimensional "vector" h_3 as follows:

$$h_3 = W_2\overline{h}_2 = [-1, 1, -3] \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} = -1$$

The output o is obtained by applying the sigmoid function to h_3. In other words, we have the following:

$$o = \frac{1}{1 + \exp(-h_3)} = \frac{1}{1+e} \approx 0.27$$

The point-specific loss is $L = -\log_e(0.27) \approx 1.3$.

Backwards phase: In the backward phase, we first start by initializing $\frac{\partial L}{\partial o}$ to $-1/o$, which is $-1/0.27$. Then, the 1-dimensional "gradient" g_3 of the hidden layer h_3 is obtained by using the backpropagation formula for the sigmoid function in Table 2.1:

$$g_3 = o(1-o) \underbrace{\frac{\partial L}{\partial o}}_{-1/o} = o - 1 = 0.27 - 1 = -0.73 \tag{2.24}$$

The gradient $\overline{g}_2$ of the hidden layer $\overline{h}_2$ is obtained by multiplying g_3 with the transpose of the weight matrix $W_2 = [-1, 1, -3]$:

$$\overline{g}_2 = W_2^T g_3 = \begin{bmatrix} -1 \\ 1 \\ -3 \end{bmatrix} (-0.73) = \begin{bmatrix} 0.73 \\ -0.73 \\ 2.19 \end{bmatrix}$$

Based on the entry in Table 2.1 for the ReLU function, the gradient $\overline{g}_2$ can be propagated backwards to $\overline{g}_1 = \frac{\partial L}{\partial \overline{h}_1}$ by copying the components of $\overline{g}_2$ to $\overline{g}_1$, when the corresponding components in $\overline{h}_1$ are positive; otherwise, the components of $\overline{g}_1$ are set to zero. Therefore, the gradient $\overline{g}_1 = \frac{\partial L}{\partial \overline{h}_1}$ can be obtained by simply copying the first and second components of $\overline{g}_2$ to the first and second components of $\overline{g}_1$, and setting the third component of $\overline{g}_1$ to 0. In other words, we have the following:

$$\overline{g}_1 = \begin{bmatrix} 0.73 \\ -0.73 \\ 0 \end{bmatrix}$$

Note that we can also compute the gradient $\overline{g}_0 = \frac{\partial L}{\partial \overline{x}}$ of the loss with respect to the input layer $\overline{x}$ by simply computing $\overline{g}_0 = W^T \overline{g}_1$. However, this is not really needed for computing loss-to-weight derivatives.

Computing loss-to-weight derivatives: So far, we have only shown how to compute loss-to-node derivatives in this particular example. These need to be converted to loss-to-weight derivatives with the additional step of multiplying with a hidden layer. Let M be the loss-to-weight derivatives for the weight matrix W between the two layers. Note that there is a one-to-one correspondence between the positions of the elements of M and W. Then, the matrix M is defined as follows:

$$M = \overline{g}_1 \overline{x}^T = \begin{bmatrix} 0.73 \\ -0.73 \\ 0 \end{bmatrix} [2, 1, 2] = \begin{bmatrix} 1.46 & 0.73 & 1.46 \\ -1.46 & -0.73 & -1.46 \\ 0 & 0 & 0 \end{bmatrix}$$

Similarly, one can compute the loss-to-weight derivative matrix M_2 for the 1×3 matrix W_2 between $\overline{h}_2$ and h_3:

$$M_2 = g_3 \overline{h}_2^T = (-0.73)[2, 1, 0] = [-1.46, -0.73, 0]$$

Note that the size of the matrix M_2 is identical to that of W_2, although the weights of the missing edges should not be updated.

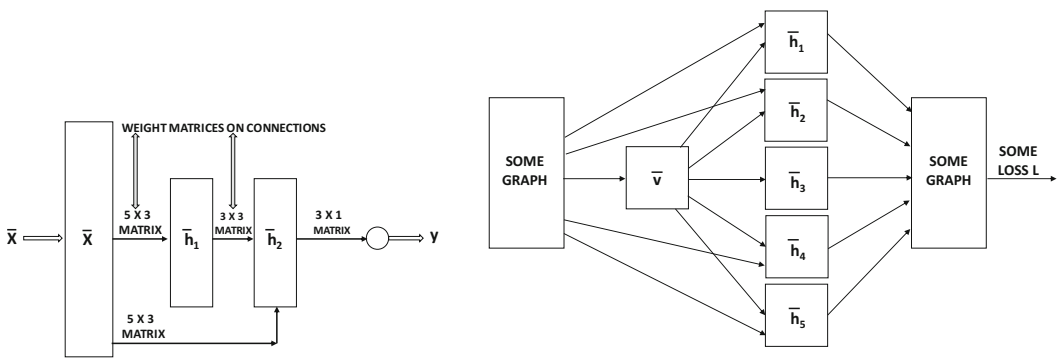
2.5.4 Vector-Centric Backpropagation with Non-Layered Architectures

Most neural networks are layered architectures in which all nodes in layer i are connected to those from layer-$(i+1)$. As a result, the computational graph appears as a single path. What happens when the computational graph has an arbitrary structure? For example, the computational graph in Figure 2.15(a) has a *skip connection* between alternate layers. This type of situation does occur in some recent convolutional architectures like *ResNet* [193, 194]. In such a case, we might have a situation where we have multiple nodes $\overline{h}_1 \ldots \overline{h}_s$ between node $\overline{v}$ and a network in later layers, as shown in Figure 2.15(b). Assume that the vector $\overline{h}_i$ has dimensionality m_i. In such a case, the partial derivative turns out to be a simple generalization of the case used for a single path:

$$\frac{\partial L}{\partial \overline{v}} = \sum_{i=1}^{s} \underbrace{\frac{\partial \overline{h}_i}{\partial \overline{v}}}_{d \times m_i} \underbrace{\frac{\partial L}{\partial \overline{h}_i}}_{m_i \times 1} = \sum_{i=1}^{s} \text{Jacobian}(\overline{h}_i, \overline{v})^T \frac{\partial L}{\partial \overline{h}_i}$$

In most layered neural networks, we only have a single path and we rarely have to deal with the case of branches. Such branches might, however, arise in the case of neural networks with skip connections (see Figures 2.15(a) and (b)). However, even in these types of complicated network architectures, *each node only has to worry about its local outgoing edges during backpropagation*. This is the key point of the backpropagation paradigm, where one does not have to worry about the global structure of the graph during gradient computation.

Consider the case where we have p output nodes containing vector-valued variables, which have indices denoted by $t_1 \ldots t_p$, and the variables in it are $\overline{y}(t_1) \ldots \overline{y}(t_p)$. In such a case, the loss function L might be function of all the components in these vectors. Assume that the ith node contains a column vector of variables denoted by $\overline{y}(i)$. Furthermore, in the denominator layout of matrix calculus, each $\overline{\Delta}(i) = \frac{\partial L}{\partial \overline{y}(i)}$ is a column vector with dimensionality equal to that of $\overline{y}(i)$. It is this *vector* of loss derivatives that will be propagated backwards. The vector-centric algorithm for computing derivatives is as follows:



(a) An unconventional architecture (b) Vector-centric graph with multiple paths

Figure 2.15: Examples of vector-centric computational graphs

Initialize $\overline{\Delta}(t_k) = \frac{\partial L}{\partial \overline{y}(t_k)}$ for each output node t_k for $k \in \{1 \ldots p\}$;
repeat
  Select an unprocessed node i such that the values of $\overline{\Delta}(j)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $\overline{\Delta}(i) \Leftarrow \sum_{j \in A(i)} \text{Jacobian}(\overline{y}(j), \overline{y}(i))^T \overline{\Delta}(j)$;
until all nodes have been selected;
for the vector $\overline{w}_i$ of edges incoming to each node i **do** compute $\frac{\partial L}{\partial \overline{w}_i} = \frac{\partial \overline{y}(i)}{\partial \overline{w}_i} \overline{\Delta}(i)$

In the final step of the above pseudocode, the derivative of vector $\overline{y}(i)$ with respect to the vector $\overline{w}_i$ is computed, which is itself the transpose of a Jacobian matrix. This final step converts a vector of partial derivatives with respect to node variables into a vector of partial derivatives with respect to weights incoming at a node.

2.6 The Not-So-Unimportant Details

In this section, we will discuss some important special cases and details of the backpropagation algorithm.

2.6.1 Mini-Batch Stochastic Gradient Descent

So far, all updates to the weights of a neural network are performed in point-specific fashion, which is referred to as *stochastic* gradient descent. In this section, we provide a justification for this choice along with related variants like *mini-batch stochastic gradient descent*. We also provide an understanding of the advantages and disadvantages of various choices.

Most machine learning problems can be recast as optimization problems over a *linearly additive sum* of the loss functions on the individual training data points. However, one rarely uses all the points at a single time, and one might *stochastically* select a point in order to update the parameters to reduce that point-specific loss. In a neural network, this process is natural because the simple methods we have introduced so far process the points one at a time in forwards and backwards propagation in order to iteratively minimize point-specific losses. However, the loss over the *entire data set* is really defined as the sum of the losses over individual training points. One can write the loss function of a neural network as the sum of point-specific losses:

$$L = \sum_{i=1}^{n} L_i \tag{2.25}$$

Here, L_i is the loss contributed by the ith training point. In gradient descent, one tries to minimize the loss function of the neural network by moving the parameters along the negative direction of the gradient of the loss *over all points*. For example, in the case of the perceptron, the parameters correspond to $\overline{W} = (w_1 \ldots w_d)$. Therefore, one would try to compute the loss of the underlying objective function over all points simultaneously and perform gradient descent. Therefore, in traditional gradient descent, one would try to perform gradient-descent steps such as the following:

$$\overline{W} \Leftarrow \overline{W} - \alpha \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2} \ldots \frac{\partial L}{\partial w_d} \right) \tag{2.26}$$

This type of derivative can also be written succinctly in vector notation (i.e., matrix calculus notation):

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}} \tag{2.27}$$

Correspondingly, it is easy to show that the true update of the perception should be the following (based on Equation 2.25):

$$\overline{W} \Leftarrow \overline{W} - \frac{\partial L}{\partial \overline{W}} = \overline{W} - \sum_{i=1}^{n} \frac{\partial L_i}{\partial \overline{W}} \tag{2.28}$$

This is possible to achieve in the perceptron, but if one were to generalize the above updates to multilayer architectures, one would need to *simultaneously* run all examples forwards and backwards through the network in order to compute the gradient. Note that the intermediate activations and derivatives *for each training instance* would need to be maintained *simultaneously* over hundreds of thousands of neural network nodes. This can be exceedingly large in most practical settings. It is, therefore, impractical to simultaneously run all examples through the network to compute the gradient with respect to the *entire data set* in one shot. The point-at-a-time update introduced so far is really a *practical approximation* of the true update by updating with the use of L_i rather than L:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} \tag{2.29}$$

Assuming a random ordering of points, each update can be viewed as a probabilistic approximation of the true update, although the long-term effect of repeated updates is approximately the same. This type of probabilistic approximation is, therefore, referred to as *stochastic* gradient descent. The main advantages of stochastic gradient descent are that it is fast and memory efficient, albeit[6] at the expense of accuracy.

The main issue with stochastic gradient descent is that the point-at-a-time approach can sometimes behave in an unstable way, because individual points in the training data might be mislabeled or have other errors. However, a middle ground is possible with the use of *mini-batch stochastic gradient descent*. The idea is to use a *batch* of points in each update rather than one point. In mini-batch stochastic descent, one uses a batch $B = \{j_1 \ldots j_m\}$ of training points for the update:

$$\overline{W} \Leftarrow \overline{W} - \alpha \sum_{i \in B} \frac{\partial L_i}{\partial \overline{W}} \tag{2.30}$$

The data used in machine learning problems often have a high level of redundancy in terms of the knowledge captured by different training points, and therefore the gradient obtained from a sample of points is usually quite accurate. Furthermore, the weights are often incorrect to such a degree at the beginning of the learning process that even a small sample of points can be used to create an excellent estimate of the gradient. This observation provides a practical foundation for the success of mini-batch stochastic gradient-descent, which often exhibits the best trade-off between stability, speed, and memory requirements.

When using mini-batch stochastic gradient descent, the outputs of each layer are matrices instead of vectors, and forward propagation requires the multiplication of the weight matrix with the activation matrix. The same is true for backward propagation in which matrices of gradients are maintained. Therefore, the implementation of mini-batch stochastic gradient descent increases the memory requirements, which is a limiting factor on the size of

[6] On the other hand, stochastic gradient descent also has the indirect benefit of *regularization* (cf. Chapter 5), and therefore the degradation is often not too significant. Sometimes, it even improves over vanilla gradient descent in terms of performance on out-of-sample data. However, it can occasionally provide poor results with certain types of data sets.

the mini-batch. However, memory is rarely a limiting factor in practical terms, because one does not gain much accuracy in gradient computation by increasing the batch size beyond a few hundred points. It is common to use powers of 2 as the size of the mini-batch, because this choice often provides the best efficiency on most hardware architectures; commonly used values are 32, 64, 128, or 256. Although the use of mini-batch stochastic gradient descent is ubiquitous in neural network learning, most of this book will use a single point for the update (i.e., pure stochastic gradient descent) for simplicity in presentation.

2.6.2 Learning Rate Decay

A constant learning rate is not desirable because it poses a dilemma to the analyst. In general, it is desirable to select a high learning rate at the beginning of the algorithm, with a lower learning rate for later phases. The dilemma is as follows. A lower learning rate used early on will cause the algorithm to take too long to come even close to an optimal solution. On the other hand, a large initial learning rate will allow the algorithm to come reasonably close to a good solution at first; however, the algorithm will then oscillate around the point for a very long time, or diverge in an unstable way, if the high rate of learning is maintained. In either case, maintaining a constant learning rate is not ideal. Allowing the learning rate to decay over time can naturally achieve the desired learning-rate adjustment to avoid these challenges.

The two most common decay mechanisms are *exponential decay* and *inverse decay*, which define the learning rate α_t in terms of the initial decay rate α_0 and epoch t as follows:

$$\alpha_t = \alpha_0 \exp(-k \cdot t) \qquad \text{[Exponential Decay]}$$

$$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \qquad \text{[Inverse Decay]}$$

The parameter k controls the rate of the decay. Another approach is to use step decay in which the learning rate is reduced by a particular factor every few epochs. For example, the learning rate might be multiplied by 0.5 every 5 epochs. A common approach is to track the loss on a held-out portion of the training data set, and reduce the learning rate whenever this loss stops improving. In some cases, the analyst might even babysit the learning process, and use an implementation in which the learning rate can be changed manually depending on the progress. This type of approach can be used with simple implementations of gradient descent, although it does not address many of the other problematic issues. Therefore, more complex algorithms for gradient descent are discussed in Chapter 4.

2.6.3 Checking the Correctness of Gradient Computation

The backpropagation algorithm is quite complex, and one might occasionally check the correctness of gradient computation. This can be performed easily with the use of numerical methods. Consider a particular weight w of a randomly selected edge in the network. Let $L(w)$ be the current value of the loss. The weight of this edge is perturbed by adding a small amount $\epsilon > 0$ to it. Then, the forward algorithm is executed with this perturbed weight and the loss $L(w + \epsilon)$ is computed. Then, the partial derivative of the loss with respect to w can be shown to be the following:

$$\frac{\partial L(w)}{\partial w} \approx \frac{L(w + \epsilon) - L(w)}{\epsilon} \tag{2.31}$$

When the partial derivatives do not match closely enough, it is easy to detect that an error must have occurred in computation. One needs to perform the above estimation for

only two or three checkpoints in the training process, which is quite efficient. However, it might be advisable to perform the checking over a modest subset of the parameters at these checkpoints. In order to determine whether the gradients are "close enough," one has to account for the absolute magnitudes of these gradients with the use of *relative ratios*.

Let the backpropagation-determined derivative be denoted by G_e, and the aforementioned estimation be denoted by G_a. Then, the relative ratio ρ is defined as follows:

$$\rho = \frac{|G_e - G_a|}{|G_e + G_a|} \tag{2.32}$$

Typically, the ratio should be less than 10^{-6}, although for some activation functions like the ReLU in which sharp changes in derivatives occur at particular points, it is possible for the numerical gradient to be different from the computed gradient. In such cases, the ratio should still be less than 10^{-3}. One can use this numerical approximation to check the correctness of the gradients with respect to a subset of weights. If there are millions of parameters, then one can test a sample of the derivatives for a quick check of correctness. It is also advisable to perform this check at two or three points in the training because the checks at initialization might correspond to special cases that do not hold at later stages.

2.6.4 Regularization

As you will learn in Chapter 5, it is important to regularize the stochastic gradient-descent updates by *shrinking* the weights every time an update is made. For example, the stochastic gradient-descent update for the ith data point, as shown in Equation 2.29, is repeated here:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} \tag{2.33}$$

In regularization, an additional shrinking factor $(1 - \alpha\lambda)$ is applied to the weight vector, where $\lambda > 0$ is the regularization parameter:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) - \alpha \frac{\partial L_i}{\partial \overline{W}} \tag{2.34}$$

As discussed in Chapter 5, regularization often improves the accuracy on out-of-sample data at prediction time (although it might worsen the accuracy on training data).

2.6.5 Loss Functions on Hidden Nodes

In some forms of sparse feature learning, even the outputs of the hidden nodes have loss functions associated with them. This occurs frequently in order to encourage specific properties of the hidden layer. In such cases, the backpropagation algorithm requires only minor modifications in which the gradient flow in the backwards direction includes the losses at the hidden nodes (by treating them in a similar manner to output nodes). This can be achieved by simple aggregation of the gradient flows resulting from different losses. At a fundamental level, the backpropagation methodology remains the same.

Consider the case in which the loss function L_i^h is associated with the hidden node $h(i)$. Furthermore, let $\Delta(i)$ denote the gradient flow from all nodes reachable from i. Then, the standard pre-activation based update is modified to add the effect of hidden node losses:

$$\Delta(i) \Leftarrow [\Phi_i' \sum_{j \in A(i)} w_{ij} \Delta(j)] + \Phi_i' \frac{\partial L_i^h}{\partial h(i)} \tag{2.35}$$

Note that the second term in the above update accounts for the effect of the loss on the hidden node i. This term simply adds to the gradient flow during backpropagation. This addition has a similar algebraic form to the value of the initialization of the output nodes. Therefore, the overall framework of backpropagation remains almost identical, with the main difference being that the backpropagation algorithm picks up additional contributions from the losses at the hidden nodes.

2.6.6 Backpropagation Tricks for Handling Shared Weights

A very common approach for regularizing neural networks is to use *shared weights*. The basic idea is that if one has some semantic insight that a similar function will be computed in different nodes of the network, then the weights associated with those nodes will be constrained to be the same value. Some examples are as follows:

1. In an *autoencoder* simulating *singular value decomposition* (cf. section 3.4.1.2 of Chapter 3), the weights in the input layer and the output layer are shared.
2. In a recurrent neural network for text (cf. Chapter 8), the weights in different temporal layers are shared, because each word in a sentence shares the same *language model*.
3. In a convolutional neural network, the same weight *filters* are used over the entire image input (cf. Chapter 9), as pixels are interpreted in a spatially consistent way.

Sharing weights in a semantically insightful way is one of the key tricks to successful neural network design. When one can identify particular sets of nodes sharing similar semantic relationships, it makes sense to use shared weights.

At first glance, the computation of the loss gradient with respect to the shared weights might seem to be an onerous task because of the complexity of loss functions in computational graphs. However, a key trick can be borrowed from differential calculus to enable simple computation. Let w be a shared weight across T nodes in the network, and the corresponding weight copies be denoted by $w_1 \ldots w_T$. Let the loss function be L. Then, it is easy to use the chain rule to show the following:

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{T} \frac{\partial L}{\partial w_i} \cdot \underbrace{\frac{\partial w_i}{\partial w}}_{=1} = \sum_{i=1}^{T} \frac{\partial L}{\partial w_i}$$

In other words, all we have to do is to pretend that these weights are independent, compute their derivatives, and add them! Therefore, we simply have to execute the backpropagation algorithm without any change and then sum up the gradients of different copies of the shared weight. This simple observation forms the basis of learning algorithms in many key neural network architectures such as recurrent, convolutional, and graph neural networks.

2.7 Tuning and Preprocessing

There are several important issues associated with the setup of the neural network, preprocessing, and initialization. First, the *hyperparameters* of the neural network (such as the learning rates and regularization parameters) need to be selected. Feature preprocessing and initialization can also be rather important. Neural networks tend to have larger parameter spaces compared to other machine learning algorithms, which magnifies the effect of preprocessing and initialization in many ways. In the following, we will discuss the basic methods

used for feature preprocessing and initialization. Strictly speaking, advanced methods like *pretraining* can also be considered initialization techniques. However, the discussion of this technique is deferred to the next chapter because it requires a deeper understanding of the model generalization issues associated with neural network training.

2.7.1 Tuning Hyperparameters

Neural networks have a large number of *hyperparameters* such as the learning rate, the weight of regularization, and so on. The term "hyperparameter" is used to specifically refer to the parameters regulating the design of the model (like learning rate and regularization), and they are different from the more fundamental parameters representing the weights of connections in the neural network. In Bayesian statistics, the notion of hyperparameter is used to control the prior distribution, although we use this definition in a somewhat loose sense here. In a sense, there is a two-tiered organization of parameters in the neural network, in which primary model parameters like weights are optimized with backpropagation only after fixing the hyperparameters either manually or with the use of a *tuning* phase. As we will discuss in section 5.3 of Chapter 5, the hyperparameters should not be tuned using the same data used for gradient descent. Rather, a portion of the data is held out as validation data, and the performance of the algorithm is tested on the validation set with various choices of hyperparameters. This type of approach ensures that the tuning process does not overfit to the training data set (while providing poor test data performance).

How should the candidate hyperparameters be selected for testing? The most well-known technique is *grid search*, in which a set of values is selected for each hyperparameter. In the most straightforward implementation of grid search, all combinations of selected values of the hyperparameters are tested in order to determine the optimal choice. One issue with this procedure is that the number of hyperparameters might be large, and the number of points in the grid increases *exponentially* with the number of hyperparameters. For example, if we have 5 hyperparameters, and we test 10 values for each hyperparameter, the training procedure needs to be executed $10^5 = 100000$ times to test its accuracy. Although one does not run such testing procedures to completion, the number of runs is still too large to be reasonably executed for most settings of even modest size. Therefore, a commonly used trick is to first work with coarse grids. Later, when one narrows down to a particular range of interest, finer grids are used. One must be careful when the optimal hyperparameter selected is at the edge of a grid range, because one would need to test beyond the range to see if better values exist.

The testing approach may at times be too expensive even with the coarse-to-fine-grained process. It has been pointed out [37] that grid-based hyperparameter exploration is not necessarily the best choice. In some cases, it makes sense to randomly sample the hyperparameters uniformly within the grid range. As in the case of grid ranges, one can perform multi-resolution sampling, where one first samples in the full grid range. One then creates a new set of grid ranges that are geometrically smaller than the previous grid ranges and centered around the optimal parameters from the previously explored samples. Sampling is repeated on this smaller box and the entire process is iteratively repeated multiple times to refine the parameters.

Another key point about sampling many types of hyperparameters is that the *logarithms* of the hyperparameters are sampled uniformly rather than the hyperparameters themselves. Two examples of such parameters include the regularization rate and the learning rate. For example, instead of sampling the learning rate α between 0.1 and 0.001, we first sample $\log(\alpha)$ uniformly between -1 and -3, and then exponentiate it to the power of 10. It is

more common to search for hyperparameters in the logarithmic space, although there are some hyperparameters that should be searched for on a uniform scale.

Finally, a key point about large-scale settings is that it is sometimes impossible to run these algorithms to completion because of the large training times involved. For example, a single run of a convolutional neural network in image processing might take a couple of weeks. Trying to run the algorithm over many different choices of parameter combinations is impractical. However, one can often obtain a reasonable estimate of the broader behavior of the algorithm in a short time. Therefore, the algorithms are often run for a certain number of epochs to test the progress. Runs that are obviously poor or diverge from convergence can be quickly killed. In many cases, multiple threads of the process with different hyperparameters can be run, and one can successively terminate or add new sampled runs. In the end, only one winner is allowed to train to completion. Sometimes a few winners may be allowed to train to completion, and their predictions will be averaged as an ensemble.

The general problem of searching for optimal neural network configurations is referred to as *Neural Architecture Search*, which has a much wider scope than what is discussed in this chapter [115]. Examples include *reinforcement learning* (cf. section 11.8.5) for architectural search or *Bayesian optimization* [41, 317] for parameter search. For smaller networks, it is possible to use libraries such as *Hyperopt* [637], *Spearmint* [639], and *SMAC* [638].

2.7.2 Feature Preprocessing

The feature processing methods used for neural network training are not very different from those in other machine learning algorithms. There are two forms of feature preprocessing used in machine learning algorithms:

1. *Additive preprocessing and mean-centering:* In many algorithms, it can be useful to mean-center the data in order to remove certain types of bias effects. Many algorithms in traditional machine learning (such as principal component analysis) also work with the assumption of mean-centered data. In such cases, a vector of column-wise means is subtracted from each data point. Mean-centering is often paired with *standardization*, which is discussed in the section of feature normalization.

   A second type of pre-processing is used when it is desired for all feature values to be non-negative. In such a case, the absolute value of the most negative entry of a feature is added to the corresponding feature value of each data point. The latter is typically combined with min-max normalization, which is discussed below.

2. *Feature normalization:* A common type of normalization is to divide each feature value by its standard deviation. When this type of feature scaling is combined with mean-centering, the data is said to have been *standardized*. The basic idea is that each feature is presumed to have been drawn from a *standard* normal distribution with zero mean and unit variance.

   The other type of feature normalization is useful when the data needs to be scaled in the range $(0, 1)$. Let min_j and max_j be the minimum and maximum values of the jth attribute. Then, each feature value x_{ij} for the jth dimension of the ith point is scaled by min-max normalization as follows:

$$x_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j} \tag{2.36}$$

Feature normalization often does ensure better performance, because it is common for the relative values of features to vary by more than an order of magnitude. In such cases, parameter learning faces the problem of ill-conditioning, in which the loss function has an inherent tendency to be more sensitive to some parameters than others. As we will see in Chapter 4, this type of ill-conditioning affects the performance of gradient descent. Therefore, it is advisable to perform the feature scaling up front.

Whitening

Another form of feature pre-processing is referred to as *whitening*, in which the axis-system is rotated to create a new set of *de-correlated* features, each of which is scaled to unit variance. Typically, principal component analysis is used to achieve this goal.

Principal component analysis can be viewed as the application of singular value decomposition *after* mean-centering a data matrix (i.e., subtracting the mean from each column). Let D be an $n \times d$ data matrix that has already been mean-centered. Let C be the $d \times d$ co-variance matrix of D in which the (i, j)th entry is the co-variance between the dimensions i and j. Because the matrix D is mean-centered, we have the following:

$$C = \frac{D^T D}{n} \propto D^T D \tag{2.37}$$

The eigenvectors of the co-variance matrix provide the de-correlated directions in the data. Furthermore, the eigenvalues provide the variance along each of the directions. Therefore, if one uses the top-k eigenvectors (i.e., largest k eigenvalues) of the covariance matrix, most of the variance in the data will be retained and the noise will be removed. One can also choose $k = d$, but this will often result in the variances along the near-zero eigenvectors being dominated by numerical errors in the computation. It is a bad idea to include dimensions in which the variance is caused by computational errors, because such dimensions will contain little useful information for learning application-specific knowledge. Furthermore, the whitening process will scale each transformed feature to unit variance, which will blow up the errors along these directions. At the very least, it is advisable to use some threshold like 10^{-5} on the magnitude of the eigenvalues. Therefore, as a practical matter, k will rarely be exactly equal to d. Alternatively, one can add 10^{-5} to each eigenvalue for regularization before scaling each dimension.

Let P be a $d \times k$ matrix in which each column contains one of the top-k eigenvectors. Then, the data matrix D can be transformed into the k-dimensional axis system by post-multiplying with the matrix P. The resulting $n \times k$ matrix U, whose rows contain the transformed k-dimensional data points, is given by the following:

$$U = DP \tag{2.38}$$

Note that the variances of the columns of U are the corresponding eigenvalues, because this is the property of the de-correlating transformation of principal component analysis. In whitening, each column of U is scaled to unit variance by dividing it with its standard deviation (i.e., the square root of the corresponding eigenvalue). The transformed features are fed into the neural network. Since whitening might reduce the number of features, this type of preprocessing might also affect the architecture of the network, because it reduces the number of inputs.

One important aspect of whitening is that one might not want to make a pass through a large data set to estimate its covariance matrix. In such cases, the covariance matrix

and columnwise means of the original data matrix can be estimated on a sample of the data. The $d \times k$ eigenvector matrix P is computed in which the columns contain the top-k eigenvectors. Subsequently, the following steps are used for each data point: (i) The mean of each column is subtracted from the corresponding feature; (ii) Each d-dimensional row vector representing a training data point (or test data point) is post-multiplied with P to create a k-dimensional row vector; (iii) Each feature of this k-dimensional representation is divided by the square-root of the corresponding eigenvalue.

The basic idea behind whitening is that data is assumed to be generated from an independent Gaussian distribution along each principal component. By whitening, one assumes that each such distribution is a *standard* normal distribution, and provides equal importance to the different features. Note that after whitening, the scatter plot of the data will roughly have a spherical shape, even if the original data is elliptically elongated with an arbitrary orientation. The idea is that the uncorrelated concepts in the data have now been scaled to equal importance (on an a priori basis), and the neural network can decide which of them to emphasize in the learning process. Another issue is that when different features are scaled very differently, the activations and gradients will be dominated by the "large" features in the initial phase of learning. This might hurt the relative learning rate of some important weights in the network.

2.7.3 Initialization

Initialization is particularly important in neural networks because of the stability issues associated with neural network training. As you will learn in section 4.3.2, neural networks often exhibit stability problems in the sense that the activations of each layer either become successively weaker or successively stronger. The effect is exponentially related to the depth of the network, and is therefore particularly severe in deep networks. One way of ameliorating this effect to some extent is to choose good initialization points in such a way that the gradients are stable across the different layers.

One possible approach to initialize the weights is to generate random values from a Gaussian distribution with zero mean and a small standard deviation, such as 10^{-2}. Typically, this will result in small random values that are both positive and negative. One problem with this initialization is that it is not sensitive to the number of inputs to a specific neuron. For example, if one neuron has only 2 inputs and another has 100 inputs, the output of the former is far more sensitive to the average weight because of the additive effect of more inputs (which will show up as a much larger gradient). In general, it can be shown that the variance of the outputs linearly scales with the number of inputs, and therefore the standard deviation scales with the square root of the number of inputs. To balance this fact, each weight is initialized to a value drawn from a normal distribution with zero mean and standard deviation $\sqrt{1/r_{in}}$, where r_{in} is the number of inputs to that neuron. Bias neurons are always initialized to zero weight. Alternatively, one can initialize the weight to a value that is uniformly distributed in $[-1/\sqrt{r_{in}}, 1/\sqrt{r_{in}}]$.

More sophisticated rules for initialization consider the fact that the nodes in different layers interact with one another to contribute to output sensitivity. Therefore a larger outdegree also contributes to an increasing magnitude of the activations. Let r_{in} and r_{out} respectively be the fan-in and fan-out for a particular neuron. One suggested initialization rule, referred to as *Xavier initialization* or *Glorot initialization* is to use a normal distribution with zero mean and standard deviation of $\sqrt{2/(r_{in} + r_{out})}$.

An important consideration in using randomized methods is that *symmetry breaking* is important. if all weights are initialized to the same value (such as 0), all updates will move

in lock-step in a layer. As a result, identical features will be created by the neurons in a layer. It is important to have a source of asymmetry among the neurons to begin with.

2.8 Backpropagation Is Interpretable

One of the common refrains about neural networks has been their lack of interpretability [99]. However, it turns out that one can use backpropagation in order to determine the features that contribute the most to the classification of a particular test instance. This provides the analyst with an understanding of the relevance of each feature to classification. This approach also has the useful property that it can be used for feature selection [422]. Consider a test instance $\overline{X} = [x_1, \ldots x_d]$, for which the multilabel output scores of the neural network are $o_1 \ldots o_k$. Furthermore, let the output of the winning class among the k outputs be o_m, where $m \in \{1 \ldots k\}$. The goal is to identify the features that are most sensitive to the classification of this test instance. Therefore, for each attribute x_i, we would like to determine the sensitivity of the output o_m to x_i, which amounts to computing the absolute magnitude of $\frac{\partial o_m}{\partial x_i}$. The sign of this derivative also tells us whether increasing x_i slightly from its current value increases or decreases the score of the winning class. For classes other than the winning class, the derivative also provides some understanding of the sensitivity, but this additional sensitivity is secondary in importance to that of the winning class, particularly when the number of classes is large. The value of $\frac{\partial o_m}{\partial x_i}$ can be computed with a straightforward application of node-to-node derivative computation (cf. section 2.3.1) all the way to the input layer rather than focusing to loss-to-weight derivatives.

One can also use this approach for feature selection by aggregating the absolute value of the gradient over all classes and all correctly classified training instances. The features with the largest aggregate sensitivity over the whole training data are the most relevant. Strictly speaking, one does not need to aggregate this value over all classes, but one can simply use only the winning class for correctly classified training instances. However, the original work in [422] aggregates this value over all classes and all instances.

These types of interpretation become particularly intuitive in computer vision [482] (see section 9.5.1 of Chapter 9), because the visual effects are sometimes spectacular. For example, for an image of a dog, the analysis will tell us which features (i.e., pixels) result in the image being considered a dog. As a result, we can create a black-and-white *saliency image* in which the portion corresponding to a dog is emphasized in light color against a dark background (cf. Figure 9.13 of Chapter 9). Backpropagation can also be used to determine the sensitivity of hidden features with respect to the original data features. For any given hidden feature h_m, the value of $\partial h_m / \partial x_i$ tells us how sensitive the hidden feature might be to a particular input. For some types of data like image data, this sensitivity can be plotted on the pixels, which provides a high level of understanding of how features are engineered. For example, each hidden feature often corresponds to a frequently occurring shape, which becomes successively complex in later layers (cf. Figure 9.16 in Chapter 9).

2.9 Summary

This chapter introduces the backpropagation algorithm in detail. A general view of backpropagation is discussed in the context of computational graphs. Subsequently, this general view is followed up by a discussion of how it is used in neural networks. Backpropagation only provides the gradient steps that are needed for effective learning. This algorithm

needs to be paired with computational algorithms for descent. Many of these algorithms are discussed in Chapter 4.

2.10 Bibliographic Notes and Software Resources

The original idea of backpropagation was based on idea of differentiation of composition of functions as developed in control theory [54, 247] under the ambit of *automatic differentiation*. The adaptation of these methods to neural networks was proposed by Paul Werbos in his PhD thesis in 1974 [543], although a more modern form of the algorithm was proposed by Rumelhart *et al.* in 1986 [424]. A discussion of the history of the backpropagation algorithm may be found in the book by Paul Werbos [544].

A discussion of algorithms for hyperparameter optimization in neural networks and other machine learning algorithms may be found in [36, 38, 507]. The random search method for hyperparameter optimization is discussed in [37]. The use of *Bayesian optimization* for hyperparameter tuning is discussed in [41, 317, 478]. Numerous libraries are available for Bayesian tuning such as *Hyperopt* [637], *Spearmint* [639], and *SMAC* [638].

The rule that the initial weights should depend on both the fan-in and fan-out of a node in proportion to $\sqrt{2/(r_{in} + r_{out})}$ is based on [147]. The analysis of initialization methods for rectifier neural networks is provided in [193]. Evaluations and analysis of the effect of feature preprocessing on neural network learning may be found in [284, 551]. The use of rectifier linear units for addressing some of the training challenges is discussed in [148].

Software Resources

The backpropagation algorithm is supported by almost all deep learning frameworks like *Caffe* [596], *Torch* [597], *Theano* [598], and *TensorFlow* [599]. *Caffe* is also available in Python and MATLAB. All these frameworks provide numerous optimization algorithms that are discussed in Chapter 4.

2.11 Exercises

1. Consider the following recurrence:

$$(x_{t+1}, y_{t+1}) = (f(x_t, y_t), g(x_t, y_t)) \tag{2.39}$$

Here, $f()$ and $g()$ are multivariate functions.

(a) Derive an expression for $\frac{\partial x_{t+2}}{\partial x_t}$ in terms of only x_t and y_t.

(b) Can you draw an architecture of a neural network corresponding to the above recursion for t varying from 1 to 5? Assume that the neurons can compute any function you want.

2. Consider a two-input neuron that multiplies its two inputs x_1 and x_2 to obtain the output o. Let L be the loss function that is computed at o. Suppose that you know that $\frac{\partial L}{\partial o} = 5$, $x_1 = 2$, and $x_2 = 3$. Compute the values of $\frac{\partial L}{\partial x_1}$ and $\frac{\partial L}{\partial x_2}$.

3. Consider a neural network with three layers including an input layer. The first (input) layer has four inputs x_1, x_2, x_3, and x_4. The second layer has six hidden units corresponding to all pairwise multiplications. The output node o simply adds the values

in the six hidden units. Let L be the loss at the output node. Suppose that you know that $\frac{\partial L}{\partial o} = 2$, and $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, and $x_4 = 4$. Compute $\frac{\partial L}{\partial x_i}$ for each i.

4. How does your answer to the previous question change when the output o is computed as a maximum of its six inputs rather than its sum?

5. The chapter discusses (cf. Table 2.1) how one can perform a backpropagation of an arbitrary function by using the multiplication with the Jacobian matrix. Discuss why one must be careful in using this matrix-centric approach.[Hint: Compute the Jacobian with respect to sigmoid function]

6. Consider the loss function $L = x^2 + y^{10}$. Implement a simple steepest-descent algorithm to plot the coordinates as they vary from the initialization point to the optimal value of 0. Consider two different initialization points of $(0.5, 0.5)$ and $(2, 2)$ and plot the trajectories in the two cases at a constant learning rate. What do you observe about the behavior of the algorithm in the two cases?

7. Consider the softmax activation function in the output layer, in which real-valued outputs $v_1 \ldots v_k$ are converted into probabilities as follows (according to Equation 2.17):

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

(a) Show that the value of $\frac{\partial o_i}{\partial v_j}$ is $o_i(1 - o_i)$ when $i = j$. In the case that $i \neq j$, show that this value is $-o_i o_j$.

(b) Use the above result to show the correctness of Equation 2.19:

$$\frac{\partial L}{\partial v_i} = o_i - y_i$$

Assume that we are using the cross-entropy loss $L = -\sum_{i=1}^{k} y_i \log(o_i)$, where $y_i \in \{0, 1\}$ is the one-hot encoded class label over different values of $i \in \{1 \ldots k\}$.

8. Consider a variation of neural architecture in Figure 2.12(b) in which a skip-connection exists between the input $\overline{x}$ and hidden layer $\overline{h}_2$ with recurrence equations as follows:

$$\overline{h}_1 = \text{ReLU}(W_1 \overline{x}), \quad \overline{h}_2 = \text{ReLU}(W_2 \overline{x} + W_3 \overline{h}_1), \quad y = W_4 \overline{h}_2$$

Here, W_1, W_2, W_3, and W_4 are matrices of appropriate size. Use the vector-centric backpropagation algorithm to derive the expressions for $\frac{\partial y}{\partial \overline{h}_2}$, $\frac{\partial y}{\partial \overline{h}_1}$, and $\frac{\partial y}{\partial \overline{x}}$ in terms of the matrices and activation values in intermediate layers.

9. Interpretability: Consider a neural network for a binary classification problem in which the output o (created by a sigmoid activation) indicates the probability that the class label is $+1$. Discuss how you can use backpropagation to find which hidden units contribute heavily to the classification of a particular test instance.

10. Let $f(x)$ be defined as follows:

$$f(x) = \sin(x) + \cos(x)$$

Consider the function $f(f(f(f(x))))$. Write this function in closed form to obtain an appreciation of the awkwardly long function. Evaluate the derivative of this function at $x = \pi/3$ radians by using a computational graph abstraction.

11. Consider a multilayer neural network with positive inputs in which each activation function produces a positive output (e.g., sigmoid). Show that all weights feeding into the same neuron will increase or decrease together in a gradient descent step. [This is one of the reasons that tanh has better convergence properties than the sigmoid.]

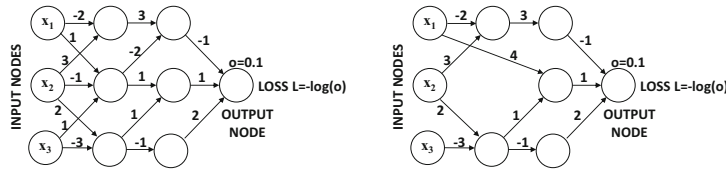
12. Forward Mode Differentiation: The backpropagation algorithm needs to compute node-to-node derivatives of *output* nodes with respect to all other nodes, and therefore computing gradients in the backwards direction makes sense (see pseudocode on page 38). However, one can also compute the node-to-node derivatives $\frac{\partial x}{\partial s_i}$ of each node x with respect to *source* (input) nodes $s_1 \ldots s_k$. Propose a variation of the pseudocode of page 38 that computes node-to-node gradients in the forward direction.

13. All-pairs node-to-node derivatives: Let $y(i)$ be the variable in node i in a directed acyclic computational graph containing n nodes and m edges. Consider the case where one wants to compute $S(i, j) = \frac{\partial y(j)}{\partial y(i)}$ for all pairs of nodes in a computational graph, so that at least one directed path exists from node i to node j. Propose an $O(n^2 m)$ algorithm for all-pairs derivative computation. [Hint: The pathwise aggregation lemma is helpful. First compute $S(i, j, t)$, which is the portion of $S(i, j)$ in the lemma belonging to paths of length exactly t, and set up an iterative relationship for increasing t.]

14. Use the pathwise aggregation lemma to compute the derivative of $y(10)$ with respect to each of $y(1)$, $y(2)$, and $y(3)$ as an algebraic expression (cf. Figure 2.7). You should get the same derivative as obtained in the text of the chapter.

15. Consider the computational graph of Figure 2.6. For a particular numerical input $x = a$, you find the unusual situation that the value $\frac{\partial y(j)}{\partial y(i)}$ is 0.3 for each and every edge (i, j) in the network. Compute the numerical value of the partial derivative of the output with respect to the input x (at $x = a$). Show the computations using both the pathwise aggregation lemma and the backpropagation algorithm.

16. Consider the computational graph of Figure 2.6. The upper node in each layer computes $\sin(x + y)$ and the lower node in each layer computes $\cos(x + y)$ with respect to its two inputs. For the first hidden layer, there is only a single input x, and therefore the values $\sin(x)$ and $\cos(x)$ are computed. The final output node computes the product of its two inputs. The single input x is 1 radian. Compute the numerical value of the partial derivative of the output with respect to the input x (at $x = 1$ radian). Show computations using both pathwise aggregation and the backpropagation.



(a) Exercise 17 (b) Exercise 18

Figure 2.16: Computational graphs for Exercises 17 and 18

17. Consider the computational graph shown in Figure 2.16(a), in which the local derivative $\frac{\partial y(j)}{\partial y(i)}$ is shown for each edge (i, j), where $y(k)$ denotes the activation of node k. The output o is 0.1, and the loss L is given by $-\log(o)$. Compute the value of $\frac{\partial L}{\partial x_i}$ for each input x_i using both path-wise aggregation and backpropagation.

18. Consider the computational graph shown in Figure 2.16(b), in which the local derivative $\frac{\partial y(j)}{\partial y(i)}$ is shown for each edge (i, j), where $y(k)$ denotes the activation of node k. The output o is 0.1, and the loss L is given by $-\log(o)$. Compute the value of $\frac{\partial L}{\partial x_i}$ for each input x_i using both path-wise aggregation and backpropagation.

Chapter 3

Machine Learning with Shallow Neural Networks

"Simplicity is the ultimate sophistication."– Leonardo da Vinci

3.1 Introduction

Conventional machine learning often uses optimization and gradient-descent methods for learning parameterized models. Neural networks are also parameterized models that are learned with continuous optimization methods. In all these cases, the machine learning model constructs a loss function in closed form, and gradient descent is used in order to learn the optimal parameters. This chapter will show that a wide variety of optimization-centric methods in machine learning can be captured with *very simple* neural network architectures containing one or two layers. In fact, neural networks can be viewed as more powerful generalizations of these simple models, in which the depth creates a loss function that is difficult to express in closed form, but is also more powerful in being to learn rich structure from the data. It is useful to show these parallels early on, as this allows the understanding of the design of a deep network as a composition of the basic units that one often uses in machine learning. Furthermore, showing this relationship provides an appreciation of the specific way in which traditional machine learning is different from neural networks, and of the cases in which one can hope to do better with neural networks.

Complex neural architectures are often an overkill in instances where data availability is limited. Additionally, it is easier to optimize traditional machine learning models in data-lean settings. On the other hand, neural networks have an increasing advantage with more data because they retain the flexibility to model more complex functions where warranted. Figure 3.1 illustrates this point in which it is shown that the advantage of neural networks increases with data availability.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_3

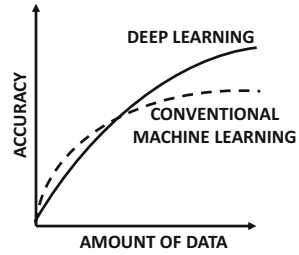


Figure 3.1: Re-visiting Figure 1.2: The effect of increased data availability on accuracy.

One way of viewing deep learning models is as a network connecting simpler computational units like the perceptron. Two examples of such simpler models are *linear regression* and *logistic regression*. A computational graph of these types of units obviously looks like a composition of the prediction functions used in simpler machine learning models. Although many neural networks can be viewed in this way, this point of view does not fully capture the complexity and the style of thinking involved in deep learning models. For example, several models (such as recurrent neural networks or convolutional neural networks) perform this stacking in a particular way with a *domain-specific understanding* of the input data. The ability to put together the basic units in a clever way is a key architectural skill required by practitioners in deep learning. Nevertheless, it is also important to learn the properties of the basic models in machine learning, since they are used repeatedly in deep learning as elementary units of computation. This chapter will, therefore, explore these basic models.

This chapter will primarily discuss two classes of models for machine learning:

1. *Supervised models:* The supervised models discussed in this chapter primarily correspond to linear models and their variants. These include methods like least-squares regression, support vector machines, and logistic regression. Multiclass variants of these models will also be studied.

2. *Unsupervised models:* The unsupervised models discussed in this chapter primarily correspond to dimensionality reduction and matrix factorization.

This chapter assumes that the reader has a basic familiarity with the classical machine learning models. Nevertheless, a brief overview of each model will also be provided to the uninitiated reader.

Chapter Organization

The next section will discuss some basic models for classification and regression, such as least-squares regression, binary Fisher discriminant, support vector machine, and logistic regression. The multiway variants of these models will be discussed in section 3.3. The use of *autoencoders* for unsupervised learning is discussed in section 3.4. Several domain-specific applications are also discussed, such as recommender systems (cf. section 3.5), text embeddings (cf. section 3.6), and graph embeddings (cf. section 3.7). A summary is given in section 3.8.

3.2 Neural Architectures for Binary Classification Models

In this section, we will discuss some basic architectures for machine learning models such as least-squares regression and classification. As we will see, the corresponding neural architectures are minor variations of the perceptron model in machine learning. The main difference is in the choice of the activation function used in the final layer, and the loss function used on these outputs. This will be a recurring theme throughout this chapter, where we will see that small changes in neural architectures can result in distinct models from traditional machine learning.

Throughout this section, we will work with a single-layer network with d input nodes and a single output node. The coefficients of the connections from the d input nodes to the output node are denoted by $\overline{W} = [w_1 \ldots w_d]^T$. Therefore, the weight vector is assumed to be a column vector. On the other hand, the training instances are rows of data matrices, and therefore each training instance $\overline{X}_i$ of feature variables is assumed to be a row vector. Furthermore, the bias will not be explicitly shown because it can be seamlessly modeled as the coefficient of an additional dummy input with a constant value of 1.

3.2.1 Revisiting the Perceptron

Let $[\overline{X}_i, y_i]$ be a training instance, in which the observed value y_i is predicted from the row vector $\overline{X}_i$ of feature variables using the following relationship:

$$\hat{y}_i = \text{sign}(\overline{W} \cdot \overline{X}_i^T) \tag{3.1}$$

Here, $\overline{W}$ is the d-dimensional column vector $[w_1, \ldots w_d]^T$ learned by the perceptron. Note the circumflex on top of $\hat{y}_i$ to indicate that it is a predicted value rather than an observed value. In general, the goal of training is to ensure that the prediction $\hat{y}_i$ is as close as possible to the observed value y_i. The gradient-descent steps of the perceptron are focused on reducing the number of misclassifications, and therefore the updates are proportional to the difference $(y_i - \hat{y}_i)$ between the observed and predicted values based on the discussion in Figure 1.3 of Chapter 1:

$$\overline{W} \Leftarrow \overline{W} + \alpha(y_i - \hat{y}_i)\overline{X}_i^T \tag{3.2}$$

A gradient-descent update that is proportional to the difference between the observed and predicted values is often caused by a squared-loss function such as $(y_i - \hat{y}_i)^2$ (see next section). However, this seemingly obvious connection to squared loss is misleading because both y_i and $\hat{y}_i$ are discrete values from $\{-1, +1\}$, and the true loss function is different from the non-differentiable value $(y_i - \hat{y}_i)^2$.

The perceptron is one of the few learning models in which the loss function was reverse engineered well after the (heuristically defined) "gradient descent" updates were proposed. To obtain the *differentiable* loss function, it is helpful to rewrite the perceptron update with the use of an indicator function $I(\cdot) \in \{0, 1\}$ that takes on 1 when the misclassification condition (i.e., $y_i\hat{y}_i < 0$) in its argument is satisfied:

$$\overline{W} \Leftarrow \overline{W} + \alpha y_i \overline{X}_i^T \, [I(y_i\hat{y}_i < 0)] \tag{3.3}$$

This rewrite from Equation 3.2 to Equation 3.3 uses the fact that $y_i = (y_i - \hat{y}_i)/2$ for misclassified points when $y_i, \hat{y}_i \in \{-1, +1\}$. Furthermore, one can absorb a constant factor

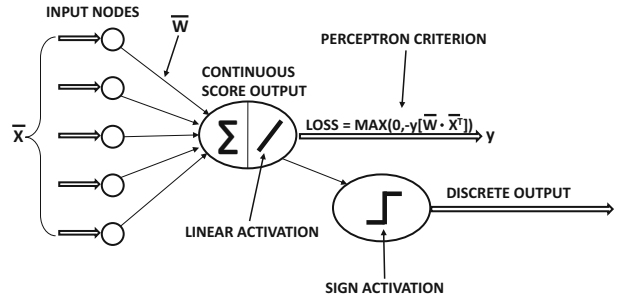


Figure 3.2: An extended architecture of the perceptron with both continuous and discrete outputs for training and prediction

of 2 within the learning rate. The update can be shown to be equivalent to gradient descent with the loss function L_i (specific to the ith training example) as follows:

$$L_i = \max\{0, -y_i \hat{y}_i\} = \max\{0, -y_i(\overline{W} \cdot \overline{X}_i^T)\} \tag{3.4}$$

This loss function is referred to as the *perceptron criterion*, which is correspondingly reflected in Figure 3.3(a). It is easy to show that the update of Equation 3.3 is a stochastic gradient-descent update:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} \tag{3.5}$$

Therefore, the perceptron updates are also stochastic gradient-descent updates, although the loss function was reverse engineered historically after the (heuristic) updates were proposed.

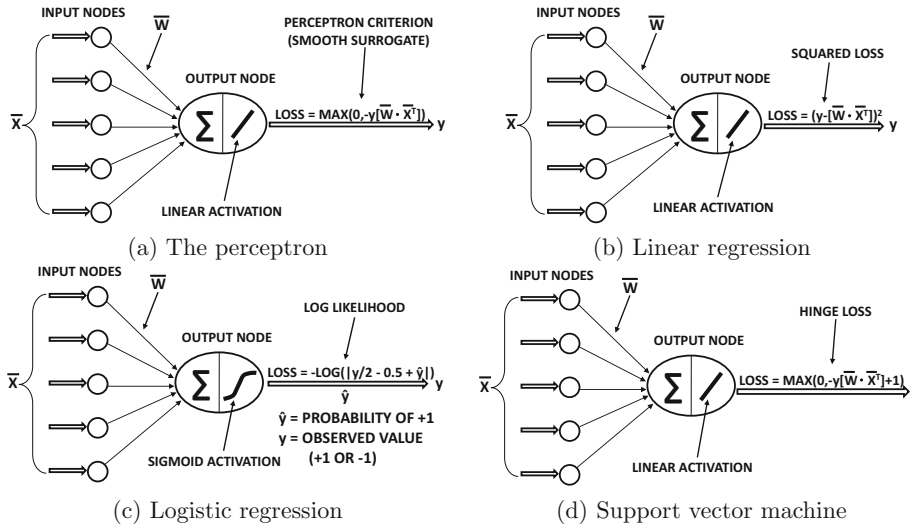
The perceptron architecture in Figure 3.3(a) uses *linear* activations to compute the continuous loss function, which is different from the sign activation used in the perceptron architecture of Chapter 1. The former is referred to as the *training architecture*, whereas the latter is referred to as the *predictive architecture*. Modern conventions on neural networks almost always depict training architectures by default. The perceptron is somewhat of an exception in often showing the predictive architecture because of the way in which its updates were initially proposed without the use of a formal loss function and heuristically defined updates with discrete outputs. One can, in fact, create an extended architecture for the perceptron (cf. Figure 3.2), in which both continuous values (for training) and discrete values (for prediction) are output. Throughout the remainder of this book, we focus on training architectures whenever an illustration of a neural network is provided.

3.2.2 Least-Squares Regression

In least-squares regression, the training data contains n different training pairs $[\overline{X}_1, y_1] \ldots [\overline{X}_n, y_n]$, where each $\overline{X}_i$ is a d-dimensional row-vector of the feature variables, and each y_i is a *real-valued* target. The fact that the target is real-valued is important, because the underlying problem is then referred to as *regression* rather than *classification*. Least-squares regression is the oldest of all learning problems, and the original formulation is owed to several nineteenth century mathematicians such as Legendre and Gauss (well before the advent of computers).

In least-squares regression, the target variable is related to the feature variables using the following relationship:

$$\hat{y}_i = \overline{W} \cdot \overline{X}_i^T \tag{3.6}$$



(a) The perceptron (b) Linear regression

(c) Logistic regression (d) Support vector machine

Figure 3.3: Basic machine learning models are small variants of one another

Note the transposition of the training instance since it is a row vector, whereas $\overline{W}$ is a column vector. Note the presence of the circumflex on top of $\hat{y}_i$ to indicate that it is a predicted value. The error of the prediction, e_i, is given by $e_i = (y_i - \hat{y}_i)$. Here, $\overline{W} = [w_1 \ldots w_d]^T$ is a d-dimensional column vector of weights that needs to be learned so as to minimize the total squared error on the training data, which is $\sum_{i=1}^{n} e_i^2$. The portion of the loss that is specific to the ith training instance is given by the following:

$$L_i = e_i^2 = (y_i - \hat{y}_i)^2 \tag{3.7}$$

This loss can be simulated with the use of an architecture similar to the perceptron except that we use squared loss instead of the perceptron criterion. This architecture is shown in Figure 3.3(b), whereas the perceptron architecture is shown in Figure 3.3(a).

As in the perceptron algorithm, the stochastic gradient-descent steps are determined by computing the gradient of e_i^2 with respect to $\overline{W}$, when the training pair $(\overline{X}_i, y_i)$ is presented to the neural network. This gradient can be computed as follows:

$$\frac{\partial e_i^2}{\partial \overline{W}} = -e_i \overline{X}_i^T \tag{3.8}$$

Therefore, the gradient-descent updates for $\overline{W}$ are computed using the above gradient and step-size α:

$$\overline{W} \Leftarrow \overline{W} + \alpha e_i \overline{X}_i^T$$

One can rewrite the above update as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha (y_i - \hat{y}_i) \overline{X}_i^T \tag{3.9}$$

the update above looks identical to the perceptron update of Equation 3.2. The updates are, however, not exactly identical because of how the predicted value $\hat{y}_i$ is computed in the two cases. In the case of the perceptron, the sign function is applied to $\overline{W} \cdot \overline{X}_i^T$ in order to compute

the binary value $\hat{y}_i$ and therefore the error $(y_i - \hat{y}_i)$ can only be drawn from $\{-2, +2\}$. In least-squares regression, the prediction $\hat{y}_i$ is a real value without the application of the sign function. This idea can also be carried on binary targets with Widrow-Hoff learning.

3.2.2.1 Widrow-Hoff Learning

One can apply least-squares regression directly to minimize the squared distance of the *real-valued* prediction $\hat{y}_i$ from the observed binary targets $y_i \in \{-1, +1\}$, and this special case of regression is referred to as *least-squares classification*. The gradient-descent update is the same as the one shown in Equation 3.9, which *looks* identical to Equation 3.2 for the perceptron. However, the least-squares classification method is different from the perceptron, because the perceptron uses $\hat{y}_i = \text{sign}\{\overline{W} \cdot \overline{X}_i^T\}$ in the update, whereas least-squares classification uses $\hat{y}_i = \overline{W} \cdot \overline{X}_i^T$. The direct application of least-squares regression to binary targets is referred to as *Widrow-Hoff learning*. The Widrow-Hoff learning rule was proposed in 1960. However, the method was not a fundamentally new one, as it is a direct application of the ancient least-squares regression method to binary targets.

The neural architecture for classification with the Widrow-Hoff method is illustrated in Figure 3.3(b). The gradient-descent steps in both the perceptron and the Widrow-Hoff method would be given by Equation 3.9, except for differences in how $(y_i - \hat{y}_i)$ is computed. In the case of the perceptron, the value $\hat{y}_i$ is an integer, and therefore the $(y_i - \hat{y}_i)$ would always be drawn from $\{-2, +2\}$. In the case of the Widrow-Hoff method, these errors can be arbitrary real values, since $\hat{y}_i$ is set to $\overline{W} \cdot \overline{X}_i^T$ without using the sign function. The learning rule of Equation 3.9, when applied to binary targets in $\{-1, +1\}$, can be alternatively referred to as least-squares classification, least mean-squares algorithm (LMS), the Widrow-Hoff learning rule, delta rule, or Adaline. The family of least-squares classification methods has been rediscovered several times in the literature under different names and with different motivations.

The loss function of the Widrow-Hoff method can also be rewritten in a different form because of its binary responses:

$$L_i = (y_i - \hat{y}_i)^2 = \underbrace{y_i^2}_{1}(y_i - \hat{y}_i)^2$$
$$= (\underbrace{y_i^2}_{1} - \hat{y}_i y_i)^2 = (1 - \hat{y}_i y_i)^2$$

This type of encoding is possible only when the target variable y_i is drawn from $\{-1, +1\}$ because we can use $y_i^2 = 1$. It is helpful to convert the Widrow-Hoff objective function to this (equivalent) form because it can be more easily related to other objective functions for binary targets (like the perceptron or the support vector machine). In fact, the term $\hat{y}_i y_i$ appears repeatedly in various loss functions for binary targets. For example, the perceptron criterion is simply $\max\{-\hat{y}_i y_i, 0\}$, whereas the loss function of the support vector machine (see next section) is $\max\{-\hat{y}_i y_i + 1, 0\}$. Almost all the binary classification models discussed in this chapter are modified versions of least-squares classification, which address some key weaknesses in the least-squares classification loss function. These weaknesses are a consequence of blindly treating binary labels as real targets, and will be discussed over the course of this chapter in the context of the better loss functions used by other algorithms.

The gradient-descent updates (cf. Equation 3.9) of least-squares regression can be rewritten slightly for Widrow-Hoff learning because of binary response variables:

$$\begin{aligned}\overline{W} &\Leftarrow \overline{W} + \alpha(y_i - \hat{y}_i)\overline{X}_i^T \quad \text{[For numeric as well as binary responses]} \\ &= \overline{W} + \alpha y_i(1 - y_i\hat{y}_i)\overline{X}_i^T \quad \text{[Only for binary responses, since } y_i^2 = 1\text{]}\end{aligned}$$

The second form of the update is helpful in relating it to perceptron, which uses an indicator function value of 1 when $1 - y_i\hat{y}_i > 0$ in lieu of $(1 - y_i\hat{y}_i)$.

3.2.2.2 Closed Form Solutions

The special case of least-squares regression and classification is solvable in closed form (without gradient-descent) by using the *pseudo-inverse* of the $n \times d$ training data matrix D, whose rows are $\overline{X}_1 \ldots \overline{X}_n$. Let the n-dimensional column vector of dependent variables be denoted by $\overline{y} = [y_1 \ldots y_n]^T$. The pseudo-inverse of matrix D is defined as follows:

$$D^+ = (D^T D)^{-1} D^T \tag{3.10}$$

Then, the optimal column-vector $\overline{W}$ can be shown [6] to be the following:

$$\overline{W} = D^+ \overline{y} \tag{3.11}$$

If regularization is incorporated, the coefficient vector $\overline{W}$ is given by the following:

$$\overline{W} = (D^T D + \lambda I)^{-1} D^T \overline{y} \tag{3.12}$$

Here, $\lambda > 0$ is the regularization parameter. However, it is more common to use gradient-descent methods like the Widrow-Hoff rule rather than using the matrix form of the solution.

3.2.3 Support Vector Machines

Consider the training data set of n instances denoted by $[\overline{X}_1, y_1]$, $[\overline{X}_2, y_2]$, $\ldots$ $[\overline{X}_n, y_n]$. Each $\overline{X}_i$ is a row vector of feature variables and y_i is the class variable drawn from $\{-1, +1\}$. The neural architecture of the support-vector machine is identical to that of least-squares classification (i.e., Widrow-Hoff method), and the only difference is in the choice of loss function. As in the case of least-squares classification, the prediction $\hat{y}_i$ for the training point $\overline{X}_i$ is obtained by applying the identity activation function on $\overline{W} \cdot \overline{X}_i^T$.

$$\hat{y}_i = \overline{W} \cdot \overline{X}_i^T$$

Here, $\overline{W} = [w_1, \ldots w_d]^T$ contains the column vector of d weights for the d different inputs into the single-layer network. Therefore, the output of the neural network is $\hat{y}_i = \overline{W} \cdot \overline{X}_i^T$ for computing the loss function, although a test instance is predicted by applying the sign function to the output.

The loss function L_i for the ith training instance in the support-vector machine is defined as follows:

$$L_i = \max\{0, 1 - y_i\hat{y}_i\} = \max\{0, 1 - y_i(\overline{W} \cdot \overline{X}_i^T)\} \tag{3.13}$$

This loss is referred to as the *hinge-loss*, and the corresponding neural architecture is illustrated in Figure 3.3(c). Note that instances in which $y_i\hat{y}_i$ is strongly positive (i.e., larger

than 1) are considered correct predictions, and are therefore not penalized by the loss function. This is different from the Widrow-Hoff loss of $(1 - y_i\hat{y}_i)^2$, which penalizes instances even when the prediction score is "confidently" correct about the true label. For example, the Widrow-Hoff loss function would heavily penalize an instance with true label $y_i = 1$ and prediction score $\hat{y}_i = +10^6$. The support vector machine does not penalize these instances. In fact, Hinton proposed the loss value of $[\max\{(1 - \hat{y}_i y_i), 0\}]^2$ to repair this problem in Widrow-Hoff loss [200], and it turned out to be identical to the loss function of the L_2-SVM (independently proposed much later).

The overall idea behind the SVM loss function is that a positive-class training instance is only penalized for the predicted score being less than 1 (i.e., the model not being correct enough in a "confident" way), and a negative-class training instance is only penalized for the prediction score being greater than -1 (i.e., not being correct enough in a "confident" way). The need to be "confident" in order to avoid penalties is referred to as the notion of *margin* in the SVM.

It is helpful to compare this loss function with the Widrow-Hoff loss value of $(1 - y_i\hat{y}_i)^2$, in which predictions are penalized for being *different* from the target values. Comparing the hinge loss of Equation 3.13 with the perceptron criterion of Equation 3.4 is even more revealing, because it suggests that the hinge loss is horizontally shifted from the perceptron criterion by one unit. Note that the perceptron does not keep the constant term of 1 on the right-hand side, whereas the hinge loss keeps this constant within the maximization function and therefore triggers updates when the model is not confident enough about correctly predicted instances. The relationship between the perceptron criterion and the hinge loss is shown in Figure 3.4. This shift is a result of the use of the notion of margin in the SVM; it is important because it gives more stability to the SVM in noisy data sets.

The stochastic gradient-descent method computes the partial derivative of the point-wise loss function L_i with respect to the elements in $\overline{W}$. The gradient is computed as follows:

$$\frac{\partial L_i}{\partial \overline{W}} = \begin{cases} -y_i \overline{X}_i^T & \text{if } y_i\hat{y}_i < 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

Therefore, the stochastic gradient method samples a point and checks whether $y_i\hat{y}_i < 1$. If this is the case, an update is performed that is proportional to $y_i \overline{X}_i^T$:

$$\overline{W} \Leftarrow \overline{W} + \alpha y_i \overline{X}_i^T \, [I(y_i\hat{y}_i < 1)] \tag{3.15}$$

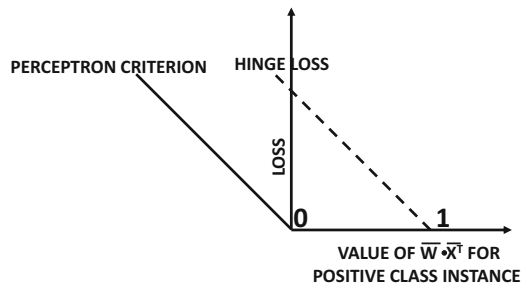


Figure 3.4: Perceptron criterion versus hinge loss

Here, $I(\cdot) \in \{0, 1\}$ is the indicator function that takes on the value of 1 when the condition in its argument is satisfied. This approach is the simplest version[1] of the primal update for SVMs [468].

The above update is *identical* to that of a perceptron (cf. Equation 3.3), except that the condition for making this update in the perceptron is $y_i \hat{y}_i < 0$, whereas the corresponding condition in the support vector machine is $y_i \hat{y}_i < 1$. Therefore, a perceptron makes the update only when a point is misclassified, whereas the support vector machine also makes updates for points that are classified correctly, albeit not very confidently. This neat relationship among their updates is a direct result of the relationships among their loss functions.

As discussed in section 1.2.2 of Chapter 1, the loss function of a perceptron has several weaknesses, which causes it to perform poorly when the data is not linearly separable. In such cases, the perceptron updates might not even converge. On the other hand, the support vector machine will usually converge, irrespective of whether the classes are linearly separable or not. For this reason, the linear support vector machine is also referred to as the *perceptron of optimal stability*.

3.2.4 Logistic Regression

Logistic regression is a probabilistic model that classifies the instances in terms of probabilities. Let $[\overline{X}_1, y_1], [\overline{X}_2, y_2], \ldots [\overline{X}_n, y_n]$ be a set of n training pairs in which $\overline{X}_i$ is a row vector containing the d-dimensional features of the ith training point and $y_i \in \{-1, +1\}$ is a binary class variable. As in the case of a perceptron, a single-layer architecture with weight vector $\overline{W} = [w_1 \ldots w_d]^T$ is used. Logistic regression applies the soft sigmoid function to $\overline{W} \cdot \overline{X}_i^T$ in order to estimate the *probability* that y_i is 1:

$$\hat{y}_i = P(y_i = 1) = \frac{1}{1 + \exp(-\overline{W} \cdot \overline{X}_i^T)} \tag{3.16}$$

Note that $P(y_i = 1)$ is 0.5 when $\overline{W} \cdot \overline{X}_i^T = 0$. Therefore, the hyperplane $\overline{W} \cdot \overline{X}_i^T = 0$ can still be considered a linear separator, although a nonzero probability of a class can be obtained on either side of the separator (with the break-even point being the separator itself). This makes logistic regression a probabilistic predictor.

We will now describe how the probabilistic training procedure. For positive samples in the training data, we want to maximize $P(y_i = 1)$ and for negative samples, we want to maximize $P(y_i = -1)$. For positive samples satisfying $y_i = 1$, one wants to maximize $\hat{y}_i$ and for negative samples satisfying $y_i = -1$, one wants to maximize $1 - \hat{y}_i$. One can write this case-wise maximization in the form of a consolidated expression of always maximizing $|y_i/2 - 0.5 + \hat{y}_i|$. The products of these probabilities must be maximized over all training instances to maximize the likelihood $\mathcal{L}$:

$$\mathcal{L} = \prod_{i=1}^{n} |y_i/2 - 0.5 + \hat{y}_i| \tag{3.17}$$

[1] In the case of the SVM, regularization is considered particularly important, although we omit it here for simplicity of comparison. The regularized update is $\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha y_i \overline{X}_i^T\ [I(y_i \hat{y}_i < 1)]$.

The goal is to maximize $\mathcal{L}$, which corresponds to a *maximum-likelihood model*. Using the negative logarithm of $\mathcal{L}$ converts the product-wise maximization is converted to additive minimization over training instances.

$$\mathcal{LL} = -\log(\mathcal{L}) = \sum_{i=1}^{n} \underbrace{-\log(|y_i/2 - 0.5 + \hat{y}_i|)}_{L_i} \tag{3.18}$$

Therefore, the loss function is set to $L_i = -\log(|y_i/2 - 0.5 + \hat{y}_i|)$ for each training instance. Additive forms of the objective function are particularly convenient for the types of stochastic gradient updates that are common in neural networks. The overall architecture and loss function is illustrated in Figure 3.3(d).

Let the loss for the ith training instance be denoted by L_i, which is also annotated in Equation 3.18. Then, the gradient of L_i with respect to the weights in $\overline{W}$ can be computed as follows:

$$\frac{\partial L_i}{\partial \overline{W}} = -\frac{\text{sign}(y_i/2 - 0.5 + \hat{y}_i)}{|y_i/2 - 0.5 + \hat{y}_i|} \cdot \frac{\partial \hat{y}_i}{\partial \overline{W}} = -\frac{\text{sign}(y_i/2 - 0.5 + \hat{y}_i)}{|y_i/2 - 0.5 + \hat{y}_i|} \cdot \frac{\exp(-\overline{W} \cdot \overline{X}_i^T)\overline{X}_i^T}{\left[1 + \exp(-\overline{W} \cdot \overline{X}_i^T)\right]^2}$$

One can substitute the two cases from $\{-1, +1\}$ for y_i and the value of $\hat{y}_i$ from Equation 3.16 to obtain the following:

$$\frac{\partial L_i}{\partial \overline{W}} = \begin{cases} -\frac{\overline{X}_i^T}{1+\exp(\overline{W} \cdot \overline{X}_i^T)} & \text{if } y_i = 1 \\ \frac{\overline{X}_i^T}{1+\exp(-\overline{W} \cdot \overline{X}_i^T)} & \text{if } y_i = -1 \end{cases}$$

Note that one can concisely write the above gradient as follows:

$$\frac{\partial L_i}{\partial \overline{W}} = -\frac{y_i \overline{X}_i^T}{1 + \exp(y_i \overline{W} \cdot \overline{X}_i^T)} = -\left[\text{Probability of mistake on } (\overline{X}_i, y_i)\right] (y_i \overline{X}_i^T) \tag{3.19}$$

Therefore, the gradient-descent updates of logistic regression are as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha \frac{y_i \overline{X}_i^T}{1 + \exp[y_i(\overline{W} \cdot \overline{X}_i^T)]} \tag{3.20}$$

Just as the perceptron and the Widrow-Hoff algorithms use (various functions of) the *magnitudes* of the mistakes to make updates, the logistic regression method uses the *probabilities* of the mistakes to make updates. This is a natural consequence of the probabilistic nature of the loss function.

It is noteworthy that the loss function of logistic regression can also be written directly in terms of the weights as follows:

$$L_i = \log(1 + \exp(-y_i[\overline{W} \cdot \overline{X}_i^T])) \tag{3.21}$$

The loss function of Equation 3.21 makes it possible to directly compare the loss functions of various models, which will be explored in the next section.

3.2.5 Comparison of Different Models

In order to explain the difference in loss functions between the perceptron, Widrow-Hoff learning, the support vector machine, and logistic regression, we list their loss functions in a table below:

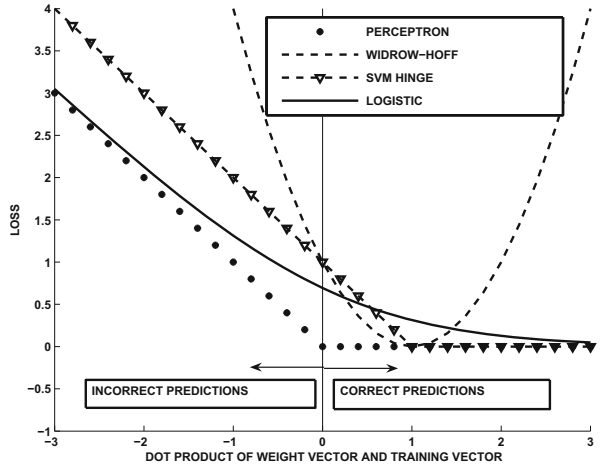


Figure 3.5: The loss functions of different variants of the perceptron for a **positive class** instance. Key observations: (i) The SVM loss is shifted from the perceptron (surrogate) loss by exactly one unit to the right; (ii) the logistic loss is a smooth variant of the SVM loss; (iii) the Widrow-Hoff loss is the only case in which points are increasingly penalized for classifying points "too correctly."

Model	**Loss function** L_i for $(\overline{X}_i, y_i)$
Perceptron	$\max\{0, -y_i \cdot (\overline{W} \cdot \overline{X}_i^T)\}$
Widrow-Hoff learning	$(y_i - \overline{W} \cdot \overline{X}_i^T)^2 = \{1 - y_i \cdot (\overline{W} \cdot \overline{X}_i^T)\}^2$
Support vector machine (Hinge)	$\max\{0, 1 - y_i \cdot (\overline{W} \cdot \overline{X}_i^T)\}$
Support vector machine (Hinton's L_2-Loss) [200]	$[\max\{0, 1 - y_i \cdot (\overline{W} \cdot \overline{X}_i^T)\}]^2$
Logistic Regression	$\log(1 + \exp[-y_i(\overline{W} \cdot \overline{X}_i^T)])$

For the case of the support vector machine, we have added Hinton's L_2-loss to the table, which has not been discussed in this chapter. This loss function is simply the square of the hinge loss, and it is historically important because it was proposed before SVMs were discovered by the machine learning community. Therefore, the L_2-SVM was proposed as an unnamed loss function in a neural architecture before it was formally proposed in traditional machine learning as the "support vector machine"! It is evident that all losses are functions of $\overline{W} \cdot \overline{X}_i^T$ and they penalize the training instance in different ways when $\overline{W} \cdot \overline{X}_i^T$ is different from the observed value y_i.

We have also shown the value of different loss functions for a *positive* training instance $[\overline{X}_i, y_i]$ at different values of $\overline{W} \cdot \overline{X}_i^T$ in Figure 3.5. The X-axis shows the value of $\overline{W} \cdot \overline{X}_i^T$ and the Y-axis shows the loss for an instance in which $y_i = +1$. Because of the fact that $y_i = +1$, the loss function should not penalize the instance less by increasing $\overline{W} \cdot \overline{X}_i^T$, and it should ideally penalize the instance very little when $\overline{W} \cdot \overline{X}_i^T \geq 1$. This is indeed the case for the support-vector machine, in which the hinge-loss function flattens out suddenly to zero loss when $\overline{W} \cdot \overline{X}_i^T$ exceeds the target value of $+1$. In other words, only misclassified points or points that are too close to the decision boundary $\overline{W} \cdot \overline{X}^T = 0$ are penalized. The perceptron criterion is identical in shape to the hinge loss, except that it is shifted by one unit to the left. Logistic regression has a loss that has a similar shape to the SVM loss

function, although it does penalize training instances slightly for values of $\overline{W} \cdot \overline{X}_i^T \geq 1$. Notably, this loss is monotonically decreasing with increasing value of $\overline{W} \cdot \overline{X}_i^T$, which is desirable for a positive-class instance. The Widrow-Hoff method is the only case in which a positive training point is penalized for having too large a positive value of $\overline{W} \cdot \overline{X}_i^T$. In other words, the Widrow-Hoff method penalizes points for being classified "too correctly." This is a potential problem with the Widrow-Hoff objective function, and many of the loss functions in machine learning (e.g., Hinton's L_2-SVM loss) were motivated by the need to repair the Widrow-Hoff loss in a one-sided way.

It is noteworthy that all the derived updates in this section typically correspond to stochastic gradient-descent updates that are encountered in traditional machine learning. The updates are the same whether or not we use a neural architecture to represent the models for these algorithms. Our main point in going through this exercise is to show that rudimentary special cases of neural networks are instantiations of well-known algorithms in the machine learning literature. The key point is that with greater availability of data one can incorporate additional nodes and depth to increase the model's power, explaining the superior behavior of neural networks with larger data sets (cf. Figure 3.1).

3.3 Neural Architectures for Multiclass Models

All the models discussed so far in this chapter are designed for binary classification. In this section, we will discuss how one can design multiway classification models by changing the architecture of the perceptron slightly, and allowing multiple output nodes.

3.3.1 Multiclass Perceptron

Consider a setting with k different classes. Therefore, each training instance $[\overline{X}_i, c(i)]$ is of the form where the value of $\overline{X}_i$ is a d-dimensional row vector of features, and $c(i) \in \{1 \ldots k\}$ is the index of the correct class for the ith training instance. In such a case, we would like to find k columns vectors of coefficients $\overline{W}_1 \ldots \overline{W}_k$ simultaneously so that the value of $\overline{W}_{c(i)} \cdot \overline{X}_i^T$ is larger than $\overline{W}_r \cdot \overline{X}_i^T$ for any $r \neq c(i)$. This is because one always predicts the training instance to the class r with the largest value of $\overline{W}_r \cdot \overline{X}_i^T$. In other words, the predicted value $hatc(i)$ of the class of the ith training instance is given by the following:

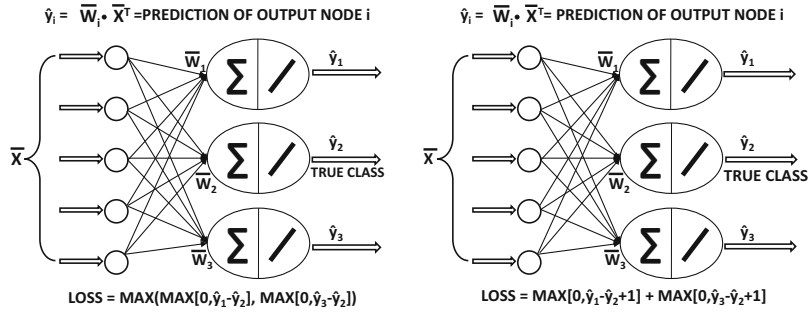
$$\hat{c}(i) = \text{argmax}_r \overline{W}_r \cdot \overline{X}_i^T$$

Note the circumflex on top of $c(i)$ indicating that it is a predicted value rather than observed value. Therefore, the loss function for the ith training instance penalizes deviation from the aforementioned desiderata on the separators as follows:

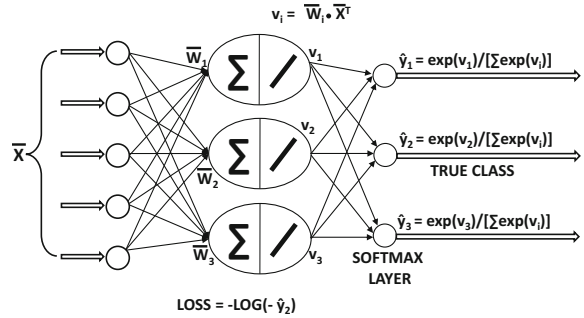
$$L_i = \max_{r:r \neq c(i)} \max(\overline{W}_r \cdot \overline{X}_i^T - \overline{W}_{c(i)} \cdot \overline{X}_i^T, 0) \tag{3.22}$$

The loss is positive whenever the wrong class $r \neq c(i)$ has the largest value of $\overline{W}_r \cdot \overline{X}_i^T$. The multiclass perceptron is illustrated in Figure 3.6(a). As in all neural network models, one can use gradient-descent in order to determine the updates. For a correctly classified instance, the gradient is always 0, and there are no updates. For a misclassified instance, the gradients are as follows:

$$\frac{\partial L_i}{\partial \overline{W}_r} = \begin{cases} -\overline{X}_i^T & \text{if } r = c(i) \\ \overline{X}_i^T & \text{if } r \neq c(i) \text{ is most misclassified prediction} \\ 0 & \text{otherwise} \end{cases} \tag{3.23}$$



(a) Multiclass perceptron (b) Multiclass SVM



(c) Multinomial logistic regression

Figure 3.6: Multiclass models: In each case, class 2 is assumed to be the ground-truth class.

Therefore, the stochastic gradient-descent method is applied as follows. Each training instance is fed into the network. If the correct class $r = c(i)$ receives the largest of output $\overline{W}_r \cdot \overline{X}_i^T$, then no update needs to be executed. Otherwise, the following update is made to each separator $\overline{W}_r$ for learning rate $\alpha > 0$:

$$\overline{W}_r \Leftarrow \overline{W}_r + \begin{cases} \alpha \overline{X}_i^T & \text{if } r = c(i) \\ -\alpha \overline{X}_i^T & \text{if } r \neq c(i) \text{ is most misclassified prediction} \\ 0 & \text{otherwise} \end{cases} \tag{3.24}$$

Only two of the separators are always updated at a given time. In the special case that $k = 2$, these gradient updates reduce to the perceptron because both the separators $\overline{W}_1$ and $\overline{W}_2$ will be related as $\overline{W}_1 = -\overline{W}_2$ if the descent is started at $\overline{W}_1 = \overline{W}_2 = 0$. Another quirk that is specific to the unregularized perceptron is that it is possible to use a learning rate of $\alpha = 1$ without affecting the learning because the value of α only has the effect of scaling the weight when starting with $\overline{W}_j = 0$ (see Exercise 2). This property is, however, not true for other linear models in which the value of α does affect the learning.

3.3.2 Weston-Watkins SVM

As in the case of the multiclass perceptron, it is assumed that the ith training instance is denoted by $[\overline{X}_i, c(i)]$, where $\overline{X}_i$ contains the d-dimensional row vector of feature variables, and $c(i)$ contains the class index drawn from $\{1, \ldots, k\}$. One wants to learn d-dimensional column vectors $\overline{W}_1 \ldots \overline{W}_k$ representing the coefficients (ideally) satisfying the condition

the class index r with the largest value of $\overline{W}_r \cdot \overline{X}_i^T$ is predicted to be the correct class $c(i)$. The Weston-Watkins loss function [548] L_i for the ith training instance $[\overline{X}_i, c(i)]$ penalizes deviation from this ideal condition, and is defined as follows:

$$L_i = \sum_{r:r \neq c(i)} \max(\overline{W}_r \cdot \overline{X}_i^T - \overline{W}_{c(i)} \cdot \overline{X}_i^T + 1, 0) \tag{3.25}$$

The neural architecture of the Weston-Watkins SVM is illustrated in Figure 3.6(b). It is instructive to compare the objective function of the Weston-Watkins SVM (Equation 3.25) with that of the multiclass perceptron (Equation 3.22). First, for each class $r \neq c(i)$, if the prediction $\overline{W}_r \cdot \overline{X}_i^T$ lags behind that of the true class by less than a margin amount of 1, then a loss is incurred for that class. Furthermore, the losses over all such classes $r \neq c(i)$ are *added*, rather than taking the maximum of the losses. These two differences accomplish the following intuitive goals:

1. The multiclass perceptron only updates the linear separator of a class that is predicted *most* incorrectly along with the linear separator of the true class. On the other hand, the Weston-Watkins SVM updates the separator of *any* class that is predicted more favorably than the true class.

2. Not only does the Weston-Watkins SVM update the separator in the case of misclassification, but it also updates the separators in cases where an incorrect class gets a prediction that is "uncomfortably close" to the true class. This is based on the notion of margin.

The gradient of the loss function with respect to each $\overline{W}_r$ is computed as follows. In the event that the loss function L_i is 0, the gradient of the loss function is 0 as well. Therefore, no update is required. However, if the loss function is non-zero, we have either a misclassified or a "barely correct" prediction in which the second-best and best class prediction are not sufficiently distinguished by the model. In such a case, an update needs to be performed. The loss function of Equation 3.25 is created by adding up the contributions of the $(k-1)$ separators belonging to the incorrect classes. Let $\delta(r, \overline{X}_i)$ be a 0/1 indicator function, which is 1 when the rth class separator contributes positively to the loss function in Equation 3.25. In such a case, the gradient of the loss function is as follows:

$$\frac{\partial L_i}{\partial \overline{W}_r} = \begin{cases} -\overline{X}_i^T [\sum_{j \neq r} \delta(j, \overline{X}_i)] & \text{if } r = c(i) \\ \overline{X}_i^T [\delta(r, \overline{X}_i)] & \text{if } r \neq c(i) \end{cases} \tag{3.26}$$

This results in the following stochastic gradient-descent step:

$$\overline{W}_r \Leftarrow \overline{W}_r + \alpha \begin{cases} \overline{X}_i^T [\sum_{j \neq r} \delta(j, \overline{X}_i)] & \text{if } r = c(i) \\ -\overline{X}_i^T [\delta(r, \overline{X}_i)] & \text{if } r \neq c(i) \end{cases} \tag{3.27}$$

Here, the parameter α denotes the learning rate. In most cases, the weight vectors are also shrunk with a regularization parameter $\lambda > 0$. One can implement regularization by adding the shrinkage update $\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda)$ after each loss-wise update.

3.3.3 Multinomial Logistic Regression (Softmax Classifier)

Multinomial logistic regression can be considered the multi-way generalization of logistic regression, just as the Weston-Watkins SVM is the multiway generalization of the binary

SVM. As in the case of the multiclass perceptron, it is assumed that the input to the model is a training data set containing pairs of the form $[\overline{X}_i, c(i)]$, where $\overline{X}_i$ is a row vector of features, and $c(i) \in \{1 \ldots k\}$ is the index of the class of d-dimensional row vector $\overline{X}_i$. As in the case of the previous two models, the class r with the largest value of $\overline{W}_r \cdot \overline{X}_i^T$ is predicted to be the label of the data point $\overline{X}_i$. However, in this case, there is an additional probabilistic interpretation of $\overline{W}_r \cdot \overline{X}_i^T$ in terms of the posterior probability $P(r|\overline{X}_i)$ that the class r is predicted given the data point $\overline{X}_i$. This estimation can be naturally accomplished with the softmax activation function:

$$P(r|\overline{X}_i) = \frac{\exp(\overline{W}_r \cdot \overline{X}_i^T)}{\sum_{j=1}^{k} \exp(\overline{W}_j \cdot \overline{X}_i^T)} \tag{3.28}$$

Note that large values of $\overline{W}_r \cdot \overline{X}_i^T$ map to large probabilities, and the probabilities over all classes always sum to 1. The loss function L_i for the ith training instance is defined by the cross-entropy loss, which is the negative logarithm of the probability of the true class. The neural architecture of the softmax classifier is illustrated in Figure 3.6(c).

The cross-entropy loss may be expressed in terms of either the input features or in terms of the softmax pre-activation values $v_r = \overline{W}_r \cdot \overline{X}_i^T$ as follows:

$$\begin{aligned} L_i &= -\log[P(c(i)|\overline{X}_i)] = -\overline{W}_{c(i)} \cdot \overline{X}_i^T + \log[\sum_{j=1}^{k} \exp(\overline{W}_j \cdot \overline{X}_i^T)] \\ &= -v_{c(i)} + \log[\sum_{j=1}^{k} \exp(v_j)] \end{aligned}$$

Therefore, the partial derivative of L_i with respect to v_r can be computed as follows:

$$\frac{\partial L_i}{\partial v_r} = \begin{cases} -\left(1 - \frac{\exp(v_r)}{\sum_{j=1}^{k} \exp(v_j)}\right) & \text{if } r = c(i) \\ \left(\frac{\exp(v_r)}{\sum_{j=1}^{k} \exp(v_j)}\right) & \text{if } r \neq c(i) \end{cases} \tag{3.29}$$

$$= \begin{cases} -(1 - P(r|\overline{X}_i)) & \text{if } r = c(i) \\ P(r|\overline{X}_i) & \text{if } r \neq c(i) \end{cases} \tag{3.30}$$

The gradient of the loss of the ith training instance with respect to the separator of the rth class is computed by using the chain rule of differential calculus in terms of its pre-activation value $v_j = \overline{W}_j \cdot \overline{X}_i^T$:

$$\frac{\partial L_i}{\partial \overline{W}_r} = \sum_j \left(\frac{\partial L_i}{\partial v_j}\right)\left(\frac{\partial v_j}{\partial \overline{W}_r}\right) = \frac{\partial L_i}{\partial v_r} \underbrace{\frac{\partial v_r}{\partial \overline{W}_r}}_{\overline{X}_i^T} \tag{3.31}$$

In the above simplification, we used the fact that v_j has a zero gradient with respect to $\overline{W}_r$ for $j \neq r$. The value of $\frac{\partial L_i}{\partial v_r}$ in Equation 3.31 can be substituted from Equation 3.30 to obtain the following result:

$$\frac{\partial L_i}{\partial \overline{W}_r} = \begin{cases} -\overline{X}_i^T (1 - P(r|\overline{X}_i)) & \text{if } r = c(i) \\ \overline{X}_i^T P(r|\overline{X}_i) & \text{if } r \neq c(i) \end{cases} \tag{3.32}$$

Note that we have expressed the gradient indirectly using probabilities (based on Equation 3.28) both for brevity and for intuitive understanding of how the gradient is related to the probability of making different types of mistakes. Each of the terms $[1 - P(r|\overline{X}_i)]$ and $P(r|\overline{X}_i)$ is the probability of making a mistake for an instance with label $c(i)$ with respect to the predictions for the rth class. The separator for the rth class is updated as follows:

$$\overline{W}_r \Leftarrow \overline{W}_r + \alpha \begin{cases} \overline{X}_i^T (1 - P(r|\overline{X}_i)) & \text{if } r = c(i) \\ -\overline{X}_i^T P(r|\overline{X}_i) & \text{if } r \neq c(i) \end{cases} \tag{3.33}$$

Here, α is the learning rate. The softmax classifier updates all the k separators for each training instance, unlike the multiclass perceptron and the Weston-Watkins SVM, each of which updates only a small subset of separators (or no separator) for each training instance. This is a consequence of probabilistic modeling, in which the notion of correctness is not absolute.

3.4 Unsupervised Learning with Autoencoders

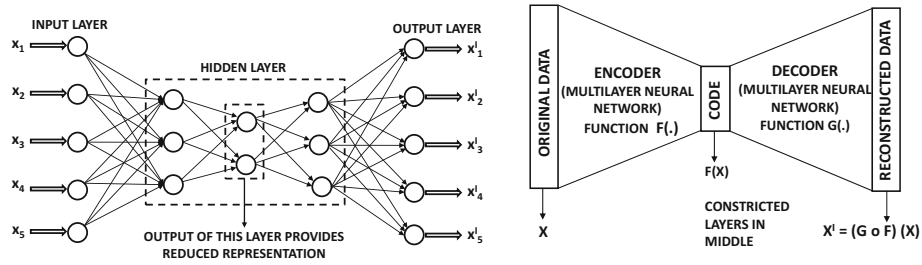
Autoencoders represent a fundamental architecture that is used for various types of unsupervised learning applications. The simplest autoencoders with linear layers map to well-known dimensionality reduction techniques like *singular value decomposition*. However, deep autoencoders with nonlinearity map to complex models that might not exist in traditional machine learning. Therefore, the goal of this section is to show two things:

1. Classical dimensionality reduction methods like singular value decomposition are special cases of shallow neural architectures.

2. By adding depth and nonlinearity to the basic architecture, one can generate sophisticated nonlinear embeddings of the data. While nonlinear embeddings are also available in traditional machine learning, the latter is limited to loss functions that can be expressed compactly in closed form. The loss functions of deep neural architectures are no longer compact; however, they provide unprecedented flexibility in controlling the properties of the embedding by making various types of architectural changes (and allowing backpropagation to take care of the complexities of differentiation).

The basic idea of an autoencoder is to have an output layer with the same dimensionality as the inputs. The idea is to try to reconstruct the input data instance. An autoencoder *replicates* the data from the input to the output, and is therefore sometimes referred to as a *replicator neural network*. Although reconstructing the data might seem like a trivial matter by simply copying the data forward from one layer to another, this is not possible when the number of units in the middle are *constricted*. In other words, the number of units in each middle layer is typically fewer than that in the input (or output). As a result, one cannot simply copy the data from one layer to another. Therefore, the activations in the constricted layers hold a reduced representation of the data; the net effect is that this type of reconstruction is inherently *lossy*. This general representation of the autoencoder is given in Figure 3.7(a), where an architecture is shown with three constricted layers. Note that the output layer has the same number of units as the input layer. The loss function of this neural network uses the sum-of-squared differences between the input and the output feature values in order to force the output to be as similar as possible to the input.

It is common (but not necessary) for an M-layer autoencoder to have a symmetric architecture between the input and output, where the number of units in the kth layer is



(a) Three hidden layers (b) General schematic

Figure 3.7: The basic schematic of the autoencoder

the same as that in the $(M-k+1)$th layer. Furthermore, the value of M is often odd, as a result of which the $(M+1)/2$th layer is the most constricted layer. The symmetry in the architecture often extends to the fact that the weights outgoing from the kth layer are tied to those incoming to the $(M-k)$th layer in many architectures. Here, we are counting the (non-computational) input layer as the first layer, and therefore the minimum number of layers in an autoencoder would be three, corresponding to the input layer, constricted layer, and the output layer.

The reduced representation of the data in the most constricted layer is also sometimes referred to as the *code*, and the number of units in this layer is the dimensionality of the reduction. The initial part of the neural architecture before the bottleneck is referred to as the *encoder* (because it creates a reduced code), and the final part of the architecture is referred to as the *decoder* (because it reconstructs from the code). The general schematic of the autoencoder is shown in Figure 3.7(b).

3.4.1 Linear Autoencoder with a Single Hidden Layer

Matrix factorization is one of the most widely studied problems in unsupervised learning, and it is used for dimensionality reduction, clustering, and predictive modeling in recommender systems. In matrix factorization, we want to factorize the $n \times d$ matrix D into an $n \times k$ matrix U and a $d \times k$ matrix V:

$$D \approx UV^T \tag{3.34}$$

Here, $k \ll n$ is the rank of the factorization. The matrix U contains the reduced representation of the data, and the matrix V contains the basis vectors. In traditional machine learning, this problem is solved by minimizing the *Frobenius norm* of the *residual matrix* denoted by $(D-UV^T)$. The squared Frobenius norm of a matrix is the sum of the squares of the entries in the matrix. Therefore, one can write the objective function of the optimization problem as follows:

$$\text{Minimize } J = \|D - UV^T\|_F^2$$

Here, the notation $\|\cdot\|_F$ indicates the Frobenius norm. The parameter matrices U and V need to be learned in order to optimize the aforementioned error. Although it is relatively easy to derive the gradient-descent steps [7] for this optimization problem (without worrying about neural networks at all), our goal here is to capture this optimization problem within a neural architecture. Going through this exercise helps us show that simple matrix factorization is a special case of an autoencoder architecture, which sets the stage for understanding the gains obtained with deeper autoencoders.

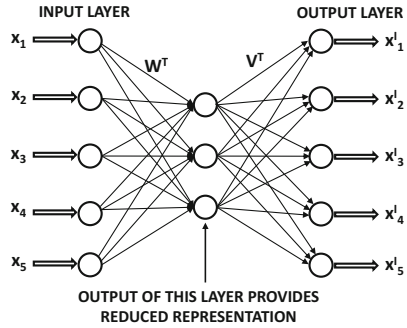


Figure 3.8: A basic autoencoder with a single layer

From the perspective of neural networks, one would need to design the architecture in such a way that the rows of D are fed to the neural network, and the reduced rows of U are obtained from the constricted layer when the original data is fed to the neural network. The single-layer autoencoder is illustrated in Figure 3.8, where the hidden layer contains k units. The rows of D are input into the autoencoder, whereas the k-dimensional rows of U are the activations of the hidden layer. Note that the $k \times d$ matrix of weights in the decoder must be V^T, since it is necessary to be able to multiply the rows of U to reconstruct the rows of $D \approx UV^T$ according to the optimization model discussed above. As shown in subsequent sections, the $d \times k$ matrix W^T of weights in the encoder can also be expressed in terms of D, U, and V.

The vector of values in a particular layer of the network can be obtained by multiplying the vector of values in the previous layer with the matrix of weights connecting the two layers (with linear activation). Here, $\overline{u}_i$ is the row vector containing the ith row of U and $\overline{X}'_i$ is the reconstruction of the ith row $\overline{X}_i$ of D. One first encodes $\overline{X}_i$ with the matrix W^T, and then decodes (reconstructs) it back to $\overline{X}'_i$ using V^T:

$$\overline{u}_i = \overline{X}_i W^T \tag{3.35}$$

$$\overline{X}'_i = \overline{u}_i V^T \tag{3.36}$$

Although neural network activations have been chosen to be column vectors throughout the book, we have used (equivalent) row-vector variants here (and forward propagation matrix operations) in order to show the similarity with traditional matrix factorization models. For example, it is not difficult to see that Equation 3.36 is a row-wise variant of the matrix-centric reconstruction in Equation 3.34. The autoencoder minimizes the sum-of-squared differences between the input and the output, which is equivalent to minimizing $\sum_i \|\overline{X}_i - \overline{u}_i V^T\|^2$. The latter is precisely the row-wise variant of the objective $\|D - UV^T\|_F^2$ of matrix factorization. Therefore, the neural architecture has exactly the same loss function as matrix factorization. However, the rows $\overline{u}_i$ in the neural network are generated as linear transformations of D with matrix W, which will cause specific relationships (i.e., constraints) to exist between D, $\overline{u}_i$ and V. It needs to be shown that this constraint does not affect the nature of the optimal solution. In the next section, we examine the nature of the linear transformation W.

Deriving Encoder Weights

As shown in Figure 3.8, the encoder weights are contained in the $k \times d$ matrix denoted by W. How is this matrix related to V? Note that the reduced representation $\overline{u}_i$ of the ith training instance $\overline{X}_i$ can be obtained by multiplying the input layer with W^T. Therefore, we have $\overline{u}_i = \overline{X}_i W^T$. We need to compute a closed-form solution to W assuming *fixed* V. Since the neural network minimizes $\sum_i \|\overline{X}_i - \overline{X}_i'\|^2$, the problem boils down to minimizing $\sum_i \|\overline{X}_i - \overline{X}_i W^T V^T\|^2$ over fixed $\overline{X}_i$ and V. The optimal solution to this problem can be shown to be $W = V^+$, where $V^+ = (V^T V)^{-1} V^T$ is the pseudoinverse of V (see Exercise 14). In other words, the optimal activation $\overline{u}_i$ is obtained by multiplying each row-vector $\overline{X}_i$ with the transposed pseudo-inverse V^+ of V:

$$\overline{u}_i = \overline{X}_i [V^+]^T$$

Therefore, the matrix U of neural network activations is related to the data set and weights as $U = D[V^+]^T$. It is also well known in traditional matrix factorization (see [6]) that if a matrix D is factorized to UV^T with least-squares loss, then the optimal matrices U and V are related by $U = D[V^+]^T$. Therefore, all objective functions and relationships between parameters and neural network activations are exactly the same as those in traditional matrix factorization.

3.4.1.1 Connections with Singular Value Decomposition

The single-layer autoencoder architecture is closely connected to *singular value decomposition (SVD)*. The loss function $\|D - UV^T\|_F^2$ of unconstrained matrix factorization is identical to that of SVD (and therefore this neural network) is identical to that of singular value decomposition (see, for example, [6]). The main difference is that the matrix factorization objective function has an infinite number of global optima, although SVD only corresponds to the specific global optimum in which the columns of V are orthonormal and those of U are mutually orthogonal (and such an alternate optimal solution always exists for the unconstrained problem [6]). The L_2-normalized columns of U and V are referred to as *left and right singular vectors*, respectively. One cannot guarantee these types of properties of U and V when using the gradient-descent updates with the neural network. Nevertheless, the subspace spanned by the k columns of the matrix V found by the neural network will be the same as that spanned by the top-k basis vectors of SVD as long as the backpropagation algorithm works perfectly. The optimal solution V can be related to the optimal solution V_{svd} of SVD as $V = V_{svd} A$ for *some arbitrary* $k \times k$ *invertible matrix* A. The corresponding matrix U is related to U_{svd} as $U = U_{svd}(A^T)^{-1}$. One can easily confirm that $UV^T = U_{svd} V_{svd}^T$, and therefore the quality of reconstruction remains unaffected. The choice of A depends on the specifics of the learning algorithm, gradient descent, or parameter initialization. One cannot easily control A using the vanilla architecture of Figure 3.8.

3.4.1.2 Sharing Weights in the Encoder and Decoder

A common practice that is used in the autoencoder construction is to share some of the weights between the encoder and the decoder. This is also referred to as *tying the weights*. In particular, the autoencoder has an inherently symmetric structure, in which the weights of the encoder and decoder are forced to be the same in symmetrically matching layers. In the single-layer case discussed above, the encoder and decoder weights are shared as follows:

$$W = V^T \tag{3.37}$$

In other words, the $d \times k$ matrix V of weights is first used to transform the d-dimensional data point $\overline{X}$ into a k-dimensional representation. Then, the matrix V^T of weights is used to reconstruct the data to its original representation.

The tying of the weights effectively means that one is minimizing $\|D - DVV^T\|_F^2$ over the entire data set D. Even though the tying of weights makes the optimization problem more constrained, *the optimal objective function value is not worsened at least in the case of this simple architecture*. However, this desirable property might not hold in the case in more complex architectures with nonlinear activation functions. One can also show that the optimal V found by gradient descent has orthonormal columns when D has at least k linearly independent columns (see Exercise 14). This brings it closer to SVD in terms of which of the alternate optima are found. The main difference from SVD is that the optimal V found by this architecture can be a k-dimensional rotreflection $V_{svd}P$ of the optimal basis matrix V_{svd} of SVD (using *any* $k \times k$ orthogonal matrix P depending on the vagaries of gradient descent). Furthermore, the $n \times k$ matrix U obtained by stacking the activations $\overline{u}_1 \ldots \overline{u}_n$ will be related to U_{svd} as $U = U_{svd}P$, and we will always have $UV^T = U_{svd}V_{svd}^T$. The columns of U may not be mutually orthogonal.

How does one tie weights when there are multiple layers? In general, one would have an odd number of hidden layers and an even number of weight matrices. It is a common practice to match up the weight matrices in a symmetric way about the middle. In such a case, the symmetrically arranged hidden layers would need to have the same numbers of units. Even though it is not necessary to share weights between the encoder and decoder portions of the architecture, it reduces the number of parameters by a factor of 2. This is beneficial from the point of view of creating a more stable model that better reconstructs out-of-sample data (even though the in-sample reconstructions might worsen slightly).

The sharing of weights does require some changes to the backpropagation algorithm during training. However, these modifications are not very difficult. All that one has to do is to perform normal backpropagation by pretending that the weights are not tied in order to compute the gradients. Then, the gradients across different copies of the same weight are added in order to compute the gradient-descent steps. The logic for handing shared weights in this way is discussed in section 2.6.6 of Chapter 2.

3.4.2 Nonlinear Activation Functions and Depth

The single-layer autoencoder does not seem to achieve much because many off-the-shelf tools exist for singular value decomposition. However, the autoencoder can be made to achieve more complex goals by using nonlinear activations and multiple layers. For example, consider a situation in which the matrix D is binary. In such a case, one can use the same neural architecture as shown in Figure 3.8, but one can also use a sigmoid function in the final layer to predict the output. This sigmoid layer is combined with negative log loss. Therefore, for a binary matrix $B = [b_{ij}]$, the model assumes the following:

$$B \sim \text{sigmoid}(UV^T) \tag{3.38}$$

Here, the sigmoid function is applied in element-wise fashion. Note the use of $\sim$ instead of $\approx$ in the above expression, which indicates that the binary matrix B is an instantiation of random draws from Bernoulli distributions with corresponding parameters contained in sigmoid(UV^T). The resulting factorization can be shown to be equivalent to *logistic matrix factorization*. The basic idea is that the (i, j)th element of UV^T is the parameter of a Bernoulli distribution, and the binary entry b_{ij} is generated from a Bernoulli distribution with these parameters. Therefore, U and V are learned using the log-likelihood loss of this

generative model. The log-likelihood loss implicitly tries to find parameter matrices U and V so that the probability of the matrix B being generated by these parameters is maximized.

Logistic matrix factorization has only recently been proposed [235] as a sophisticated matrix factorization method for binary data, which is useful for recommender systems with *implicit feedback* ratings. Implicit feedback refers to the binary actions of users such as buying or not buying specific items. The solution methodology of this recent work on logistic matrix factorization [235] seems to be vastly different from SVD, and it is not based on a neural network approach. However, for a neural network practitioner, the change from the SVD model to that of logistic matrix factorization is a relatively small one, where only the final layer of the neural network needs to be changed. It is this modular nature of neural networks that makes them so attractive to engineers and encourages all types of experimentation. This benefit multiplies as one moves to deeper architectures.

The real power of autoencoders in the neural network domain is realized when deeper variants are used. For example, an autoencoder with three hidden layers is shown in Figure 3.7(a). One can increase the number of intermediate layers in order to further increase the representation power of the neural network. It is noteworthy that it is essential for some of the layers of the deep autoencoder to use a nonlinear activation function to increase its representation power. As shown in Lemma 1.5.1 of Chapter 1, no additional power is gained by a multilayer network when only linear activations are used. Although this result was shown in Chapter 1 for the classification problem, it is broadly true for any type of multilayer neural network (including an autoencoder).

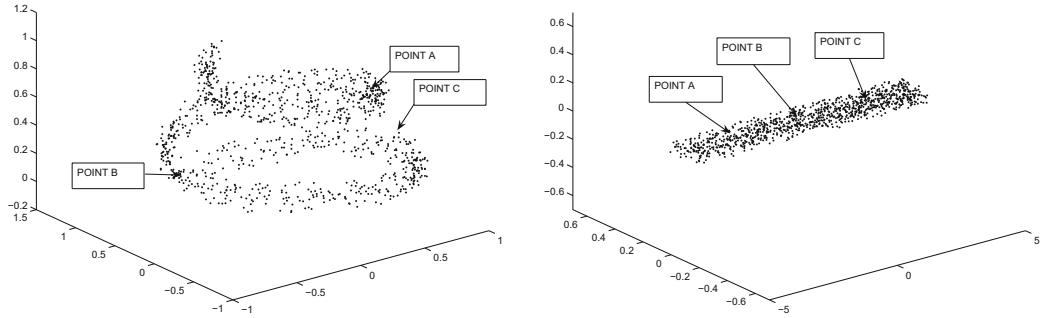
Deep networks with multiple layers provide an extraordinary amount of representation power. The multiple layers of this network provide *hierarchically* reduced representations of the data. For some data domains like images, hierarchically reduced representations are particularly natural. Indeed, for multilayer variants of the autoencoder, an exact counterpart often does not even exist in traditional machine learning. An important point is that even though the loss function might be too complex to express in closed form, the backpropagation approach rescues us from the challenges associated in computing the complicated gradient-descent steps.

Nonlinear dimensionality reduction methods can map the data into much lower dimensional spaces (with good reconstruction characteristics) than would be possible with methods like singular value decomposition. An example of a data set, which is distributed on a nonlinear spiral, is shown in Figure 3.9(a). This data set cannot be reduced to lower dimensionality using singular value decomposition (without causing significant reconstruction error). However, the use of nonlinear dimensionality reduction methods can flatten out the nonlinear spiral into a 2-dimensional representation. This representation is shown in Figure 3.9(b).

Autoencoders form the workhorse of unsupervised learning in the neural network domain. They are used for a host of applications, which will be discussed throughout this book. After training an autoencoder, it is not necessary to use both the encoder and decoder portions. For example, when using the approach for dimensionality reduction, one can use the encoder portion in order to create the reduced representations of the data. The reconstructions of the decoder might not be required at all.

3.4.3 Application to Visualization

It is possible to achieve extraordinarily compact reductions by using this approach. Greater reduction is always achieved by using nonlinear units, which implicitly map warped manifolds into linear hyperplanes. The superior reduction in these cases is because it is easier to



(a) A nonlinear pattern in three dimensions (b) A reduced data set in two dimensions

Figure 3.9: The effect of nonlinear dimensionality reduction. This figure is drawn for illustrative purposes only.

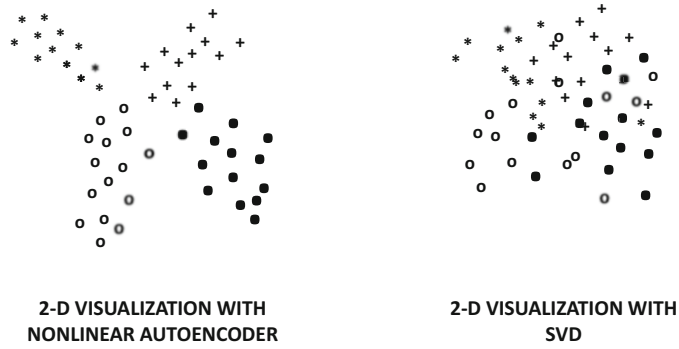


Figure 3.10: A depiction of the typical difference between the embeddings created by nonlinear autoencoders and singular value decomposition. Nonlinear and deep autoencoders are often able to separate out the entangled class structures in the underlying data, which is not possible within the constraints of linear transformations like singular value decomposition.

thread a warped surface (as opposed to a linear surface) through a larger number of points. This property of nonlinear autoencoders is often used for 2-dimensional visualizations of the data by creating a deep autoencoder in which the most compact hidden layer has only two dimensions. These two dimensions can then be mapped on a plane to visualize the points. In many cases, the class structure of the data is exposed in terms of well-separated clusters.

An illustrative example of the typical behavior of the dimensionality reduction obtained by autoencoders is shown in Figure 3.10, in which the 2-dimensional mapping created by a deep autoencoder seems to clearly separate out the different classes. On the other hand, the mapping created by singular value decomposition does not seem to separate the classes well. Figure 3.9, which provides a nonlinear spiral mapped to a linear hyperplane, clarifies the reason for this behavior. In many cases, the data may contain heavily entangled spirals (or other shapes). Linear dimensionality reduction methods cannot attain clear separation because nonlinearly entangled shapes are not linearly separable. On the other hand, deep autoencoders with nonlinearity are far more powerful and able to disentangle such shapes.

Deep autoencoders can sometimes be used as alternatives to other robust visualization methods like t-distributed stochastic neighbor embedding (t-SNE) [316]. The advantage of an autoencoder over t-SNE is that it is easier to generalize to out-of-sample data. When new data points are received, they can simply be passed through the encoder portion of the autoencoder in order to add them to the current set of visualized points.

3.4.4 Application to Outlier Detection

Dimensionality reduction is closely related to outlier detection, because outlier points are hard to encode and decode without losing substantial information. It is a well-known fact that if a matrix D is factorized as $D \approx D' = UV^T$, then the low-rank matrix D' is a de-noised representative of the data. After all, the compressed representation U captures only the regularities in the data, and is unable to capture the unusual variations in specific points. As a result, reconstruction to D' misses all these unusual variations.

The absolute values of the entries of $(D - D')$ represent the outlier scores of the matrix entries. Therefore, one can use this approach to find outlier entries, or add the squared scores of the entries in each row of D to find the outlier score of that row. Therefore, one can identify outlier data points. Furthermore, by adding the squared scores in each column of D, one can find outlier features. This is useful for applications like feature selection in clustering, where a feature with a large outlier score can be removed because it adds noise to the clustering. Refer to the bibliographic notes for more details of outlier detection methods.

3.4.5 Application to Multimodal Embeddings

One can also use an autoencoder for embedding multimodal data in a joint latent space. Multimodal data is essentially data in which the input features are heterogeneous. For example, an image with descriptive tags can be considered multimodal data. Multimodal data pose challenges to mining applications because different features require different types of processing and treatment. By embedding the heterogeneous attributes in a unified space, one is removing this source of difficulty in the mining process. An autoencoder can be used to embed the heterogeneous data into a joint space. An example of such a setting is shown in Figure 3.11. This figure shows an autoencoder with only a single layer, although one might have multiple layers in general [368, 484]. Such joint spaces can be very useful in a

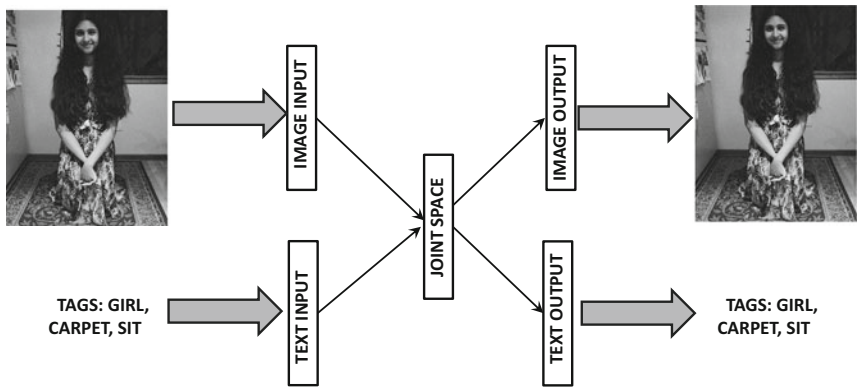


Figure 3.11: Multimodal embedding with autoencoders

variety of applications, because it often turns out to be hard to integrate data from multiple sources in known algorithms without changing the algorithms in an ad hoc manner. On the other hand, the integrated representation of the data in the hidden layer can be directly fed to any machine learning algorithm that works with multidimensional data.

3.4.6 Benefits of Autoencoders

Although several methods for nonlinear dimensionality reduction are known in machine learning, neural networks have some advantages over these methods:

1. Many nonlinear dimensionality reduction methods have a hard time mapping out-of-sample data points to reduced representations, unless these points are included in the training data up front. On the other hand, it is a simple matter to compute the reduced representation of an out-of-sample point by passing it through the network.

2. Neural networks allow more power and flexibility in the nonlinear data reduction by varying on the number and type of layers used in intermediate stages. Furthermore, by choosing specific types of activation functions in particular layers, one can engineer the nature of the reduction to the properties of the data. For example, it makes sense to use a logistic output layer with logarithmic loss for a binary data set.

From an architectural point of view, the amount of effort required by the analyst to change from one architecture to the other is often a few lines of code. This is because modern softwares for building neural networks often provide templates for describing the architecture of the neural network, where each layer is specified independently. In a sense, the neural network is "built" with well-known machine-learning units much like a child puts together building blocks of a toy. Backpropagation takes care of the details of optimization, while shielding the user from the complexities of the steps. Consider the significant mathematical differences between the specific details of SVD and logistic matrix factorization. Changing the output layer from linear to sigmoid (along with a change of loss function) can literally be a matter of changing a trivially small number of lines of code without touching most of the remaining code. This type of modularity is tremendously useful in application-centric settings.

3.5 Recommender Systems

One of the most interesting applications of matrix factorization is the design of neural architectures for recommender systems. Consider an $n \times d$ ratings matrix D with n users and d items. The (i, j)th entry of the matrix is the rating of user i for item j. However, most entries in the matrix are not observed (i.e., missing), which creates difficulties in using a traditional autoencoder architecture. This is because traditional autoencoders are designed for fully specified matrices, in which a single *row* of the matrix is input at one time. On the other hand, recommender systems are inherently suited to *elementwise* learning, in which a very small subset of ratings from a row may be available. As a practical matter, one might consider the input to a recommender system as a set of triplets of the following form:

$$\langle \text{RowId} \rangle, \langle \text{ ColumnId } \rangle, \langle \text{ Rating } \rangle$$

Our goal is to learn a k-dimensional parameter vector $\overline{u}_i$ for each user $i \in \{1 \ldots n\}$, and a k-dimensional parameter vector $\overline{v}_j$ for item $j \in \{1 \ldots d\}$. One would like to learn

these parameters so that $\overline{u}_i \cdot \overline{v}_j$ yields an approximation of the (i, j)th entry of the ratings matrix. Note that this is the same condition as traditional matrix factorization $D \approx UV^T$ by treating $\overline{u}_i$ and $\overline{v}_j$ as rows of U and V, respectively. However, the difference between recommender systems and traditional matrix factorization is that the matrix D is not fully specified; therefore, one must learn each $\overline{u}_i$ and $\overline{v}_j$ using triplet-centric input. This makes it difficult to use the traditional autoencoder architecture that is used in (fully specified) matrix factorization.

Therefore, a different type of neural architecture is used in which a one-hot encoded row index is mapped to the ratings in that row. The input layer contains n input units, which is the same as the number of rows (users). The input is a *one-hot encoded* index of the row identifier; a one-hot encoding refers to the fact that one of the n inputs is 1 and the remaining take on the value of 0. Therefore, the input uniquely specifies one of the n users. The hidden layer contains k units, where k is the rank of the factorization. Finally, the output layer contains d real-valued units with identity activation, where d is the number of columns (items). The output is a vector containing the d ratings (even though only a small subset of them are observed). All layers of the network are densely connected. The goal is to train the neural network by using the rows of the data matrix D as training points. For each row of the data matrix, the one-hot encoded row index is input, and the network outputs all the ratings of that user. Furthermore, even though the network outputs all the ratings of that user, the ground truth information of only a subset of the ratings is available. Therefore, the loss function needs to account for this missing information.

It turns out that *the weights of the neural network encode the parameters in the matrices U and V required to reconstruct $D \approx UV^T$.* Consider a setting in which the $n \times k$ input-to-hidden matrix is U, and the $k \times d$ hidden-to-output matrix is V^T. The entries of the matrix U are denoted by u_{iq}, and those of the matrix V are denoted by v_{jq}. The k weights from the ith user (input) to the k hidden units create the ith row of U, which is denoted by $\overline{u}_i = [u_{i1}, \ldots, u_{ik}]$. The k weights from the k hidden units to the jth item (output) create the jth row of V, which is denoted by $\overline{v}_j = [v_{j1} \ldots v_{jd}]$. All layers use identity activation, and therefore there is no nonlinearity in this architecture. The loss function is the sum of the squares of the errors in the output layer. However, because of the missing entries, not all output nodes have an observed output value, and the updates are performed only with respect to entries that are known. The overall architecture of this neural network is illustrated in Figure 3.12. For any particular row-wise input we are training on a neural network that is a subset of this base network, depending on which entries are specified.

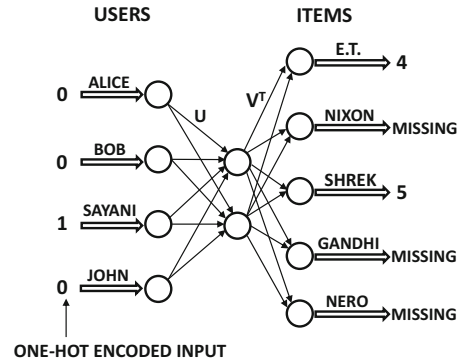


Figure 3.12: Row-index-to-value encoder for matrix factorization with missing values.

However, it is possible to give *predictions* for all outputs in the network (even though a loss function cannot be computed for missing entries).

How can we show that the aforementioned neural architecture performs matrix factorization? Consider a situation in which we are trying to reconstruct the ratings for the rth user using the neural network. Then, the input to the network is the one-hot encoded input vector $\overline{e}_r$ of length n, in which only the rth entry takes on the value of 1 and the remaining take on the value of 0. Assume that $\overline{e}_r$ is a row vector. Because of this input, the neural network simply copies the weights in $\overline{u}_r$ to the hidden units, which can also be written as $\overline{e}_r U$. Furthermore, the next layer of weights contains the matrix V^T, and therefore vector of d outputs is obtained by multiplying the hidden activations $\overline{e}_r U$ with V^T, which is $\overline{e}_r U V^T$. One can also understand this process in the context of the fact that linear layers in a network perform matrix multiplication, and this neural network contains U and V^T as the weight matrices. In essence, when the rth user is input to the neural network as a one-hot encoded vector, the computations in the neural network pull out the rth row from the product UV^T of the weight matrices U and V^T in the neural network. The corresponding d values in the rth row of UV^T appear at the output layer and represent the item-wise ratings predictions for the rth user. Therefore, all feature values are reconstructed in one shot.

How is training performed? The main attraction of this architecture is that one can perform the training either in row-wise fashion or in element-wise fashion. When performing the training in row-wise fashion, the one-hot encoded index for that row is input, and all *specified* entries of that row are used to compute the loss. The backpropagation algorithm is executed by aggregated the squared loss *only at output nodes where the ratings are specified*. From a theoretical point of view, each row is being trained on a slightly different neural network with a subset of the base output nodes (depending on which entries are observed), although the weights for the different neural networks are shared. This situation is shown in Figure 3.13, where the neural networks for the movie ratings of two different users, Bob and Sayani, are shown. For example, Bob is missing a rating for *Shrek*, as a result of which the corresponding output node is missing. However, since both users have specified a rating for *E.T.*, the k-dimensional hidden factors for this movie in matrix V will be updated during backpropagation when either Bob or Sayani is processed.

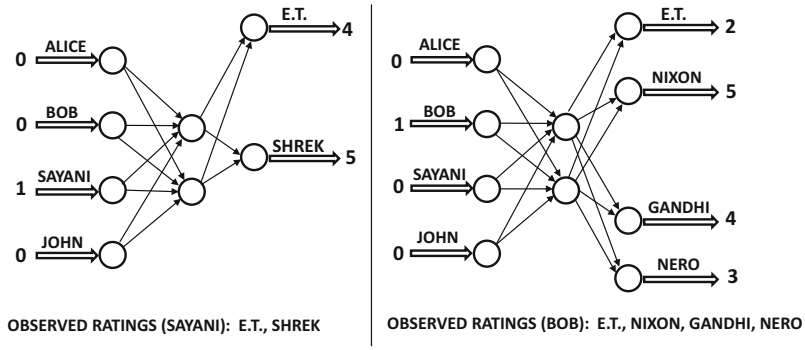


Figure 3.13: Output nodes are dropped for missing values at training time. All output nodes are materialized during prediction.

It is also possible to perform the training in element-wise fashion, where a single triplet is input. In such a case, the loss is computed only with respect to a single column index specified in the triplet. Consider the case where the row index is i, and the column index is j. In this specific case, and the single error computed at the output layer is $y - \hat{y} = e_{ij}$, which is the error in predicting rating j for user i. the backpropagation algorithm essentially updates the weights on all the k paths from node j in the output layer to the node i in the input layer. These k paths pass through the k nodes in the hidden layer. It is easy to show that the update along the qth such path is as follows:

$$u_{iq} \Leftarrow u_{iq}(1 - \alpha\lambda) + \alpha e_{ij} v_{jq}$$
$$v_{jq} \Leftarrow v_{jq}(1 - \alpha\lambda) + \alpha e_{ij} u_{iq}$$

Here, α is the step-size, and λ is the regularization parameter. *These updates are identical to those used in stochastic gradient descent for matrix factorization in recommender systems.* However, an important advantage of the use of the neural architecture (over traditional matrix factorization) is that we can vary on it in so many different ways in order to enforce different properties. For example, we can incorporate multiple hidden layers to create more powerful models when data availability is greater. The resulting loss function might not even be (compactly) expressible in closed form (and therefore outside the ambit of traditional machine learning). For matrices with binary data, we can use a logistic layer in the output. This will result in *logistic matrix factorization.* For matrices with categorical entries (and count-centric weights attached to entries), one can use a softmax layer at the very end. This will result in *multinomial matrix factorization.* To date, we are not aware of a formal description of multinomial matrix factorization in traditional machine learning; yet, it is a simple modification of the neural architecture (implicitly) used by recommender systems. In general, it is often easy to stumble upon sophisticated models when working with neural architectures because of their modular structure. The neural network abstraction brings practitioners (without too much mathematical training) much closer to sophisticated methods in machine learning, while being shielded from the details of optimization by the backpropagation framework.

3.6 Text Embedding with Word2vec

Neural network methods have been used to learn word embeddings of text data. Such embeddings are learned from word-word context matrices. For a lexicon of d words, the (i, j)th entry of such a $d \times d$ *word-word context* matrix contains the frequency of the number of times that word j occurs in the context of word i. The notion of "context" refers to a window of length t placed around the word in a sentence, and it therefore contains $2t$ words (not including the central "target" word creating the context). There are two ways in which the embedding can be generated. The first is to directly factorize the word-word context matrix. The second is to generate a neural model in which context words are target words are generated from one another by extracting windows of training points from the training sentences. As we will show in this section, *these two methods are equivalent.*

Consider a sentence containing the words $w_1 w_2 \ldots w_n$ in that sequence. One generates multiple training points from each sentence using context windows. There are two variants of *word2vec*, depending on whether target words are predicted from contexts or whether contexts are predicted from target words:

1. *Predicting target words from contexts:* This model tries to predict the ith word, w_i, in a sentence using a window of width t around the word. Therefore, the words

$w_{i-t}w_{i-t+1}\ldots w_{i-1}w_{i+1}\ldots w_{i+t-1}w_{i+t}$ are used to predict the target word w_i. This model is also referred to as the *continuous bag-of-words (CBOW) model*.

2. *Predicting contexts from target words:* This model tries to predict the context $w_{i-t}w_{i-t+1}\ldots w_{i-1}w_{i+1}\ldots w_{i+t-1}w_{i+t}$ around word w_i, given the ith word in the sentence, denoted by w_i. This model is referred to as the *skip-gram model*. There are, however, two ways in which one can perform this prediction. The first technique is a *multinomial* model which predicts one word out of d outcomes. The second model is a Bernoulli model, which predicts whether or not each context is present for a particular word. The second model uses *negative sampling* of contexts for better efficiency and accuracy.

Each of these methods will be discussed in this section.

3.6.1 Neural Embedding with Continuous Bag of Words

In the continuous bag-of-words (CBOW) model, the training pairs are all context-word pairs in which a window of context words is input, and a single target word is predicted. The context contains $2 \cdot t$ words, corresponding to t words both before and after the target word. For notational ease, we will use the length $m = 2 \cdot t$ to define the length of the context. Therefore, the input to the system is a set of m words. Without loss of generality, let the subscripts of these words be numbered so that they are denoted by $w_1 \ldots w_m$, and let the target (output) word in the middle of the context window be denoted by w. Note that w can be viewed as a categorical variable with d possible values, where d is the size of the lexicon. The neural model predicts the probability $P(w|w_1w_2\ldots w_m)$ in the output node and creates a loss function that implicitly maximizes the product of these probabilities over all training samples (by minimizing the sum of the corresponding negative log-probabilities over all training instances).

The overall architecture of this model is illustrated in Figure 3.14. In the architecture, we have a single input layer with $m \times d$ nodes, a hidden layer with p nodes, and an output layer with d nodes. The nodes in the input layer are clustered into m different groups, each of which has d units. Each group with d input units is the one-hot encoded input vector of one of the m context words being modeled by CBOW. Only one of these d inputs will be 1 and the remaining inputs will be 0. Therefore, one can represent an input x_{ij} with two indices corresponding to contextual position and word identifier. Specifically, the input $x_{ij} \in \{0, 1\}$ contains two indices i and j in the subscript, where $i \in \{1 \ldots m\}$ is the position of the context, and $j \in \{1 \ldots d\}$ is the identifier of the word.

The hidden layer contains p units, where p is the dimensionality of the hidden layer in *word2vec*. Let $h_1, h_2, \ldots h_p$ be the outputs of the hidden layer nodes. Note that each of the d words in the lexicon has m different representatives in the input layer corresponding to the m different context words, but the weight of each of these m connections is the same. Such weights are referred to as shared. Sharing weights is a common trick used for regularization in neural networks, when one has specific insight about the domain at hand. Let the shared weight of each connection from the jth word in the lexicon to the qth hidden layer node be denoted by u_{jq}. Note that each of the m groups in the input layer has connections to the hidden layer that are defined by the same $d \times p$ weight matrix U. This situation is shown in Figure 3.14.

It is noteworthy that $\overline{u}_j = [u_{j1}, u_{j2}, \ldots u_{jp}]$ (i.e., jth row of U) can be viewed as the p-dimensional embedding of the jth input word over the entire corpus, and $\overline{h} = [h_1 \ldots h_p]$ provides the embedding of a specific instantiation of an input context. Then, the output

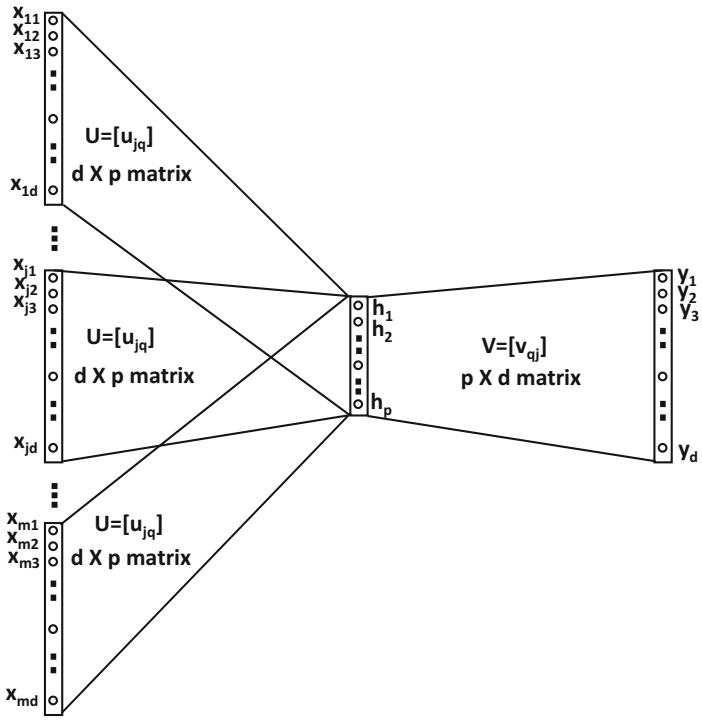


Figure 3.14: Word2vec: The CBOW model. Note the similarities and differences with Figure 3.12, which uses a single set of inputs with a linear output layer. One could equivalently choose to collapse the m sets of d one-hot encoded input nodes into a single set of d *real-valued* input nodes, which are fed the averages of the m one-hot encoded inputs to achieve the same effect.

of the hidden layer is obtained by averaging[2] the embeddings of the words present in the context. One can write this relationship in vector form:

$$\overline{h} = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{d} \overline{u}_j x_{ij} \tag{3.39}$$

In essence, the one-hot encodings of the input words are aggregated, which implies that the ordering of the words within the window of size m does not affect the output of the model. This is the reason that the model is referred to as the continuous bag-of-words model.

The embedding $[h_1 \ldots h_p]$ is used to predict the probability that the target word is one of each of the d outputs with the use of the softmax function. The weights in the output layer are parameterized with a $d \times p$ matrix $V = [v_{jq}]$. The jth row of V is denoted by $\overline{v}_j$. Since the hidden-to-output matrix is a $p \times d$ matrix, it is annotated by V^T instead of V in Figure 3.14. Note that the multiplication of the hidden vector $\overline{h}$ with V^T to create $\overline{h}V^T$ will create a real-valued vector $[\overline{h} \cdot \overline{v}_1, \ldots, \overline{h} \cdot \overline{v}_d]$. The softmax function is applied to this vector

[2]When m is fixed over all sampled windows, it is possible to omit the factor of m in the denominator on the right-hand side of Equation 3.39. This would make the transformation identical to that of a traditional linear layer.

to convert it to d probabilities $\hat{y}_1 \ldots \hat{y}_d$ summing to 1. These outputs are the d probabilities for the target word, given the context:

$$\hat{y}_j = P(y_j = 1 | w_1 \ldots w_m) = \frac{\exp(\overline{h} \cdot \overline{v}_j)}{\sum_{k=1}^{d} \exp(\overline{h} \cdot \overline{v}_k)} \tag{3.40}$$

The *ground-truth* value of only one of the outputs $y_1 \ldots y_d$ is 1 and the remaining values are 0 for a given training instance: follows:

$$y_j = \begin{cases} 1 & \text{if the target word } w \text{ is the } j \text{ th word} \\ 0 & \text{otherwise} \end{cases} \tag{3.41}$$

For a particular target word $w = r \in \{1 \ldots d\}$, the loss function is given by $L = -\log[P(y_r = 1 | w_1 \ldots w_m)] = -\log(\hat{y}_r)$. The use of the negative logarithm turns the multiplicative likelihoods over different training instances into an additive loss function using the negative log-likelihoods.

The updates on the rows of U and V are defined by computing the gradients using the backpropagation algorithm, as training instances are passed through the neural network one by one. The update equations with learning rate α are as follows:

$$\overline{u}_i \Leftarrow \overline{u}_i - \alpha \frac{\partial L}{\partial \overline{u}_i} \quad \forall i$$

$$\overline{v}_j \Leftarrow \overline{v}_j - \alpha \frac{\partial L}{\partial \overline{v}_j} \quad \forall j$$

Next, we provide expressions for the aforementioned partial derivatives of this loss function [337, 339, 420]. The probability of making a mistake in prediction on the jth word in the lexicon is defined by $|y_j - \hat{y}_j|$. However, we use *signed* mistakes ϵ_j, in which only the correct word with $y_j = 1$ is given a positive mistake value, while all the other words in the lexicon receive negative mistake values. This is achieved by dropping the modulus:

$$\epsilon_j = y_j - \hat{y}_j \tag{3.42}$$

Note that ϵ_j is also the negative derivative of the cross-entropy loss with respect to jth input into the softmax layer (cf. Equation 2.19). Further backpropagation results in the following updates:

$$\overline{u}_i \Leftarrow \overline{u}_i + \frac{\alpha}{m} \sum_{j=1}^{d} \epsilon_j \overline{v}_j \quad [\forall \text{ words } i \text{ present in context window}]$$

$$\overline{v}_j \Leftarrow \overline{v}_j + \alpha \epsilon_j \overline{h} \quad [\forall \ j \text{ in lexicon}]$$

Here, $\alpha > 0$ is the learning rate. Repetitions of the same word i in the context window trigger multiple updates of $\overline{u}_i$. The use of an aggregate context window in the input has a smoothing effect on the CBOW model, which is particularly helpful with smaller data sets.

The training examples of context-target pairs are presented one by one, and the weights are trained to convergence. After the final training, one obtains not one but two different embeddings corresponding to the p-dimensional rows of the matrix U and the p-dimensional rows of the matrix V. The former type of embedding of words is referred to as the *input* embedding, whereas the latter is referred to as the *output* embedding. In the CBOW model, the input corresponds to the context, and therefore it makes sense to use the output embedding. However, the input embedding (or the sum/concatenation of input and output embeddings) can also be helpful for many tasks.

3.6.2 Neural Embedding with Skip-Gram Model

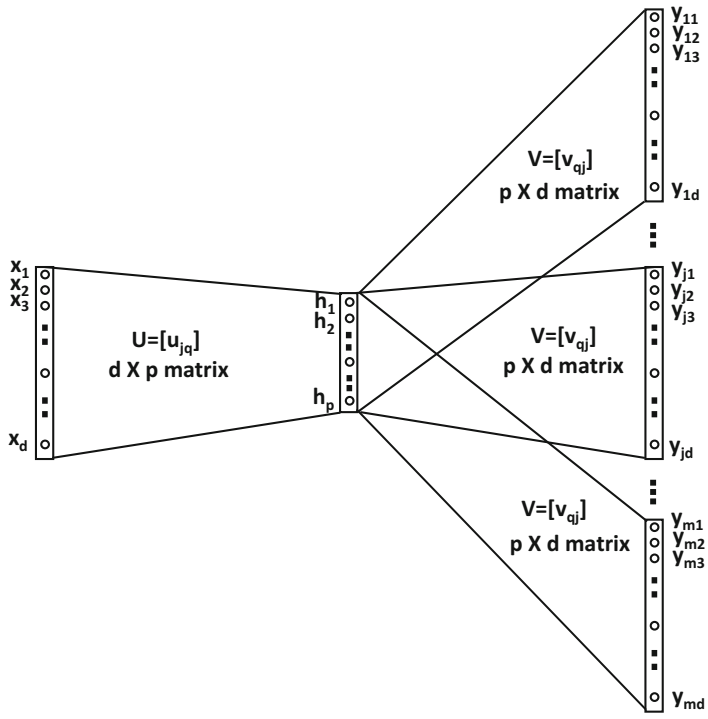
The skip-gram model uses a single target word w as the input and outputs the m context words denoted by $w_1 \ldots w_m$. Therefore, the goal is to predict $P(w_1, w_2 \ldots . w_m|w)$, which is different from the quantity $P(w|w_1 \ldots w_m)$ predicted in the CBOW model. As in the case of the continuous bag-of-words model, we can use one-hot encoding of the (categorical) input and outputs in the skip-gram model. After such an encoding, the skip-gram model will have d binary inputs denoted by $x_1 \ldots x_d$ corresponding to the d possible values of the single input word. Similarly, the output of each training instance is encoded as $m \times d$ values $y_{ij} \in \{0, 1\}$, where i ranges from 1 to m (size of context window), and j ranges from 1 to d (lexicon size). Each $y_{ij} \in \{0, 1\}$ indicates whether the ith contextual word takes on the jth possible value for that training instance. However, the (i, j)th output node only computes a soft probability value $\hat{y}_{ij} = P(y_{ij} = 1|w)$. Therefore, the probabilities $\hat{y}_{ij}$ in the output layer for fixed i and varying j sum to 1, since the ith contextual position takes on exactly one of the d words. The hidden layer contains p units denoted by the vector $\overline{h} = [h_1 \ldots h_p]$. Each input x_j is connected to all the hidden nodes with a $d \times p$ matrix U in which the ith row is denoted by $\overline{u}_i$.

Furthermore, the p hidden nodes are connected to each of the m groups of d output nodes with the same set of shared weights. This set of shared weights between the p hidden nodes and the d output nodes of each of the context words is defined by the $p \times d$ matrix V^T. Therefore, V is a $d \times p$ matrix in which the jth row is $\overline{v}_j$. Note that the input-output structure of the skip-gram model is an inverted version of the input-output structure of the CBOW model. The neural architecture of the skip-gram model is illustrated in Figure 3.15(a). However, in the case of the skip-gram model, one can collapse the m identical outputs into a single output, and achieve the same results simply by using a *particular type of mini-batching* during stochastic gradient descent. In particular, all elements of a single context window are always forced to belong to the same mini-batch. This architecture is shown in Figure 3.15(b). Since the value of m is small, this specific type of mini-batching has a very limited effect, and the simplified architecture of Figure 3.15(b) is sufficient to describe the model whether or not any specific type of mini-batching is used. For the purpose of further discussion, we will use the architecture of Figure 3.15(a).

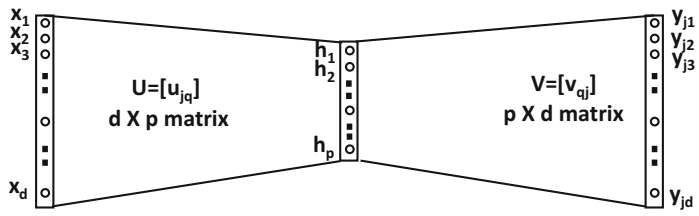
The output of the hidden layer can be computed from the input layer using the $d \times p$ matrix of weights $U = [u_{jq}]$ between the input and hidden layer as follows:

$$\overline{h} = \sum_{j=1}^{d} \overline{u}_j x_j \tag{3.43}$$

The above equation has a simple interpretation because of the one-hot encoding of the input word w in terms of $x_1 \ldots x_d$. If the input word w is the rth word, then one simply copies $\overline{u}_r$ to the hidden layer. In other words, the rth row $\overline{u}_r$ of U is copied to the hidden layer. As discussed above, the hidden layer is connected to m groups of d output nodes, each of which is connected to the hidden layer with a $d \times p$ matrix $V = [v_{jq}]$. Since the hidden-to-output matrix is of size $p \times d$, the matrix V^T is used instead of V. Each of these m groups of d output nodes computes the probabilities of the various words for a particular context word. The jth column of V is denoted by $\overline{v}_j$ and represents the output embedding of the jth word. The output $\hat{y}_{ij}$ is the probability that the word in the ith context position takes on the jth word of the lexicon. However, since the same matrix V is shared by all groups, the



(a) All elements in context window explicitly shown



MINIBATCH THE m d-DIMENSIONAL OUTPUT VECTORS IN EACH CONTEXT WINDOW DURING STOCHASTIC GRADIENT DESCENT. THE SHOWN OUTPUTS y_{jk} CORRESPOND TO THE jth OF m OUTPUTS.

(b) All elements in context window not explicitly shown

Figure 3.15: Word2vec: The skip-gram model. Note the similarity with Figure 3.12, which uses a single set of linear outputs. One could also choose to collapse the m sets of d output nodes in (a) into a single set of d outputs, and mini-batch the m instances in a single context window during stochastic gradient descent to achieve the same effect. All elements in the mini-batch are explicitly shown in (a), whereas the elements of the mini-batch are not explicitly shown in (b). However, both are equivalent as long as the nature of mini-batching is respected.

neural network predicts the same multinomial distribution for each of the context words. Therefore, we have the following:

$$\hat{y}_{ij} = P(y_{ij} = 1|w) = \underbrace{\frac{\exp(\overline{h} \cdot \overline{v}_j)}{\sum_{k=1}^{d} \exp(\overline{h} \cdot \overline{v}_k)}}_{\text{Independent of context position } i} \qquad \forall i \in \{1 \ldots m\} \tag{3.44}$$

Note that the probability $\hat{y}_{ij}$ is the same for varying i and fixed j, since the right-hand side of the above equation does not depend on the exact location i in the context window.

The loss function for the backpropagation algorithm is the negative of the log-likelihood values of the ground truth $y_{ij} \in \{0, 1\}$ of a training instance. This loss function L is given by the following:

$$L = -\sum_{i=1}^{m} \sum_{j=1}^{d} y_{ij} \log(\hat{y}_{ij}) \tag{3.45}$$

Note that the value outside the logarithm is a ground-truth binary value, whereas the value inside the logarithm is a predicted (probability) value. Since y_{ij} is one-hot encoded for fixed i and varying j, the objective function has only m non-zero terms. For each training instance, this loss function is used in combination with backpropagation to update the weights of the connections between the nodes. The update equations with learning rate α are as follows:

$$\overline{u}_i \Leftarrow \overline{u}_i - \alpha \frac{\partial L}{\partial \overline{u}_i} \quad \forall i$$

$$\overline{v}_j \Leftarrow \overline{v}_j - \alpha \frac{\partial L}{\partial \overline{v}_j} \quad \forall j$$

Next, we derive the gradients above. The probability of making a mistake in predicting the jth word in the lexicon for the ith context is defined by $|y_{ij} - \hat{y}_{ij}|$. However, we use *signed* mistakes ϵ_{ij} in which only the predicted words (positive examples) have a positive probability. This is achieved by dropping the modulus:

$$\epsilon_{ij} = y_{ij} - \hat{y}_{ij} \tag{3.46}$$

Then, the updates for a particular input word r and its output context are as follows:

$$\overline{u}_r \Leftarrow \overline{u}_r + \alpha \sum_{j=1}^{d} \left[\sum_{i=1}^{m} \epsilon_{ij} \right] \overline{v}_j \qquad \text{[Only for input word } r\text{]}$$

$$\overline{v}_j \Leftarrow \overline{v}_j + \alpha \left[\sum_{i=1}^{m} \epsilon_{ij} \right] \overline{h} \qquad \text{[For all words } j \text{ in lexicon]}$$

Here, $\alpha > 0$ is the learning rate. The p-dimensional rows of the matrix U are used as the embeddings of the words. In other words, the convention is to use the input embeddings in the rows of U rather than the output embeddings in the rows of V. It is stated in [294] that adding the input and output embeddings can help in some tasks (but hurt in others). The concatenation of the two can also be useful.

Practical Issues

Several practical issues are associated with the accuracy and efficiency of the *word2vec* framework. Increasing the embedding dimensionality improves discrimination, but it requires a greater amount of data. In general, the typical embedding dimensionality is of the

order of several hundred, although it is possible to choose dimensionalities in the thousands for very large collections. The size of the context window typically varies between 5 and 10, with larger window sizes being used for the skip-gram model as compared to the CBOW model. Using a random window size is a variant that has the implicit effect of giving greater weight to words that are placed close together. The skip-gram model is slower but it works better for infrequent words and for larger data sets.

Another issue is that the effect of frequent and less discriminative words (e.g., "*the*") can dominate the results. Therefore, a common approach is to downsample the frequent words, which improves both accuracy and efficiency. Note that downsampling frequent words has the implicit effect of increasing the context window size because dropping a word in the middle of two words brings the latter pair closer. The words that are very rare are misspellings, and it is hard to create a meaningful embedding for them without overfitting. Therefore, such words are ignored.

From a computational point of view, the updates of output embeddings are expensive. This is caused by applying the softmax over a lexicon of d words, which requires an update of each $\overline{v}_j$. One possibility is to use a *hierarchical softmax model* [337, 339, 420]. A more popular option changes the loss function (and neural architecture) slightly, which is referred to as *skipgram with negative sampling*. This approach is discussed in the next section.

Skip-Gram with Negative Sampling

In the *skip-gram with negative sampling (SGNS)* model [339], both presence or absence of word-context pairs are used for training. The basic idea is that instead of directly predicting each of the m words in the context window, we independently try to predict whether or not each of the d words in the lexicon is present in the window. In other words, the final layer of Figure 3.15 is not a softmax prediction, but a Bernoulli layer of sigmoids. The output unit for each word at each context position in Figure 3.15 is a sigmoid providing a probability value that the position takes on that word. As the ground-truth values are also available, it is possible to use the logistic loss function over all the words. Therefore, in this point of view, even the prediction problem is defined differently. Of course, it is computationally inefficient to try to make binary predictions for all d words, but the fact that the predictions of the different words are independent allows one to use only a subset of the negative outputs (since most words in the lexicon are not present). Therefore, the SGNS approach uses all the positive words in a context window and a *sample* of negative words. The number of negative samples is k times the number of positive samples. Here, k is a parameter controlling the sampling rate. Negative sampling becomes essential in this modified prediction problem to avoid learning trivial weights that predict all examples to 1. In other words, we cannot choose to avoid negative samples entirely (i.e., we cannot set $k = 0$). *This model is extremely similar to the recommender systems model with incomplete outputs*, as discussed in section 3.5. The main difference is that the "unavailability" of some of the outputs is caused by *voluntary* negative sampling rather than paucity of observed ratings (as in recommender systems).

How does one generate the negative samples? The vanilla unigram distribution samples words in proportion to their relative frequencies $f_1 \ldots f_d$ in the corpus. Better results are obtained [339] by sampling words in proportion to $f_j^{3/4}$ rather than f_j. As in all *word2vec* models, let U be a $d \times p$ matrix representing the input embedding, and V be a $d \times p$ matrix representing the output embedding. Let $\overline{u}_i$ be the p-dimensional row of U (input embedding of ith word) and $\overline{v}_j$ be the p-dimensional row of V (output embedding of jth word). Let $\mathcal{P}$ be the set of positive target-context word pairs in a context window, and $\mathcal{N}$ be the set

of negative target-context word pairs which are created by sampling. Therefore, the size of $\mathcal{P}$ is equal to the context window m, and that of $\mathcal{N}$ is $m \cdot k$. Then, the (minimization) objective function for each context window is obtained by summing up the logistic loss over the m positive samples and $m \cdot k$ negative samples:

$$O = -\sum_{(i,j)\in\mathcal{P}} \log(P[\text{Predict } (i,j) \text{ to } 1]) - \sum_{(i,j)\in\mathcal{N}} \log(P[\text{Predict } (i,j) \text{ to } 0]) \tag{3.47}$$

$$= -\sum_{(i,j)\in\mathcal{P}} \log\left(\frac{1}{1+\exp(-\overline{u}_i \cdot \overline{v}_j)}\right) - \sum_{(i,j)\in\mathcal{N}} \log\left(\frac{1}{1+\exp(\overline{u}_i \cdot \overline{v}_j)}\right) \tag{3.48}$$

This modified objective function is used in the skip-gram with negative sampling (SGNS) model in order to update the weights of U and V. SGNS is mathematically different from the basic skip-gram model discussed earlier. SGNS is not only efficient, but it also provides the best results among the different variants of skip-gram models.

What Is the Actual Neural Architecture of SGNS?

Even though the original *word2vec* paper seems to treat SGNS as an efficiency optimization of the skip-gram model, it is using a fundamentally different architecture in terms of the activation function used in the final layer. Unfortunately, the original *word2vec* paper does not explicitly point this out (and only provides the changed objective function), which causes confusion.

The modified neural architecture of SGNS is as follows. The softmax layer is no longer used in the SGNS implementation. Rather, each observed value y_{ij} in Figure 3.15 is *independently* treated as a *binary* outcome, rather than as a multinomial outcome in which the probabilistic predictions of different outcomes at a contextual position depend on one another. Instead of using softmax to create the prediction $\hat{y}_{ij}$, it uses the sigmoid activation to create probabilistic predictions $\hat{y}_{ij}$, whether each y_{ij} is 0 or 1. Then, one can add up the log loss of $\hat{y}_{ij}$ with respect to observed y_{ij} over all $m \cdot d$ possible values of (i, j) to create the full loss function of a context window. However, this is impractical because the number of zero values of y_{ij} is too large and zero values are noisy anyway. Therefore, SGNS uses negative sampling to approximate this *modified* objective function. This means that for each context window, we are backpropagating from only a subset of the $m \cdot d$ outputs in Figure 3.15. The size of this subset is $m + m \cdot k$. This is where efficiency is achieved. However, since the final layer uses binary predictions (with sigmoids), it makes the SGNS architecture fundamentally different from the vanilla skip-gram model even in terms of the basic neural network it uses (i.e., logistic instead of softmax activation). The difference between the SGNS model and the vanilla skip-gram model is analogous to the difference between the Bernoulli and multinomial models in naïve Bayes classification (with negative sampling applied only to the Bernoulli model). Obviously, one cannot be considered a direct efficiency optimization of the other.

3.6.3 Word2vec (SGNS) is Logistic Matrix Factorization

Even though the work in [293] shows an *implicit* relationship between *word2vec* and matrix factorization, we provide a more direct relationship here. The architectures of the skip-gram models look suspiciously similar to those used in row index to value prediction in recommender systems (cf. section 3.5). The use of a backpropagation from a subset of observed outputs is similar to the negative sampling idea, except that the dropping of

outputs in negative sampling is performed for the purpose of efficiency. However, unlike the linear outputs of Figure 3.12 in section 3.5, the SGNS model uses logistic outputs to model binary predictions. The SGNS model of *word2vec* can be simulated with logistic matrix factorization. To understand the similarity with the problem setting of section 3.5, one can understand the predictions of a particular word-context window using the following triplets:

$$\langle \text{WordId} \rangle, \langle \text{Context WordId} \rangle, \langle 0/1 \rangle$$

Each context window produces $m \cdot d$ such triplets, although negative sampling only uses $m \cdot k + m$ of them, and *mini-batches* them during training. This mini-batching is another source of the difference between the architectures between Figures 3.12 and 3.15, wherein the latter has m different groups of outputs to accommodate m positive samples. However, these differences are relatively superficial, and one can still use logistic matrix factorization to represent the underlying model.

Let $B = [b_{ij}]$ be a binary matrix in which the (i, j)th value is 1 if word j occurs at least once in the context of word i in the data set, and 0 otherwise. The weight c_{ij} for any word (i, j) that occurs in the corpus is defined by the number of times word j occurs in the context of word i. The weights of the zero entries in B are defined as follows. For each row i in B we sample $k \sum_j b_{ij}$ different entries from row i, among the entries for which $b_{ij} = 0$, and the frequency with which the jth word is sampled is proportional to $f_j^{3/4}$. These are the negative samples, and one sets the weights c_{ij} for the negative samples (i.e., those for which $b_{ij} = 0$) to the number of times that each entry is sampled. As in *word2vec*, the p-dimensional embeddings of the ith word and jth context are denoted by $\overline{u}_i$ and $\overline{v}_j$, respectively. The simplest way of factorizing is to use *weighted* matrix factorization of B with the Frobenius norm:

$$\text{Minimize}_{U,V} \sum_{i,j} c_{ij}(b_{ij} - \overline{u}_i \cdot \overline{v}_j)^2 \tag{3.49}$$

Even though the matrix B is of size $O(d^2)$, this matrix factorization only has a limited number of nonzero terms in the objective function, which have $c_{ij} > 0$. These weights are dependent on co-occurrence counts, but some zero entries also have positive weight. Therefore, the stochastic gradient-descent steps only have to focus on entries with $c_{ij} > 0$. Each cycle of stochastic gradient-descent is *linear* in the number of non-zero entries, as in the SGNS implementation of *word2vec*.

However, this objective function also looks somewhat different from *word2vec*, which has a logistic form. Just as it is advisable to replace linear regression with logistic regression in supervised learning of binary targets, one can use the same trick in matrix factorization of binary matrices [235]. We can change the squared error term to the familiar likelihood term L_{ij}, which is used in logistic regression:

$$L_{ij} = \left| b_{ij} - \frac{1}{1 + \exp(\overline{u}_i \cdot \overline{v}_j)} \right| \tag{3.50}$$

The value of L_{ij} always lies in the range $(0, 1)$, and higher values indicate greater likelihood (which results in a maximization objective). The modulus in the above expression flips the sign only for the negative samples in which $b_{ij} = 0$. Now, one can optimize the following objective function in minimization form:

$$\text{Minimize}_{U,V} J = -\sum_{i,j} c_{ij} \log(L_{ij}) \tag{3.51}$$

The main difference from the objective function (cf. Equation 3.48) of *word2vec* is that this is a global objective function over all matrix entries, rather than a local objective function over a particular context window. Using mini-batch stochastic gradient-descent in matrix factorization (with an appropriately chosen mini-batch) makes the approach almost identical to *word2vec*'s backpropagation updates.

How can one interpret this type of factorization? Instead of $B \approx UV$, we have $B \approx f(UV)$, where $f(\cdot)$ is the sigmoid function. More precisely, this is a *probabilistic* factorization in which one computes the product of matrices U and V, and then applies the sigmoid function to obtain the parameters of the Bernoulli distribution from which B is generated:

$$P(b_{ij} = 1) = \frac{1}{1 + \exp(-\overline{u}_i \cdot \overline{v}_j)} \qquad \text{[Matrix factorization analog of logistic regression]}$$

It is also easy to verify from Equation 3.50 that L_{ij} is $P(b_{ij} = 1)$ for positive samples and $P(b_{ij} = 0)$ for negative samples. Therefore, the objective function of the factorization is in the form of log-likelihood maximization. This type of logistic matrix factorization is commonly used [235] in recommender systems with binary data (e.g., user click-streams).

Gradient Descent

It is also helpful to examine the gradient-descent steps of the factorization. One can compute the derivative of J with respect to the input and output embeddings:

$$\begin{aligned}\frac{\partial J}{\partial \overline{u}_i} &= -\sum_{j:b_{ij}=1} \frac{c_{ij}\overline{v}_j}{1 + \exp(\overline{u}_i \cdot \overline{v}_j)} + \sum_{j:b_{ij}=0} \frac{c_{ij}\overline{v}_j}{1 + \exp(-\overline{u}_i \cdot \overline{v}_j)} \\ &= \underbrace{-\sum_{j:b_{ij}=1} c_{ij} P(b_{ij} = 0)\overline{v}_j}_{\text{Positive Mistakes}} + \underbrace{\sum_{j:b_{ij}=0} c_{ij} P(b_{ij} = 1)\overline{v}_j}_{\text{Negative Mistakes}}\end{aligned}$$

$$\begin{aligned}\frac{\partial J}{\partial \overline{v}_j} &= -\sum_{i:b_{ij}=1} \frac{c_{ij}\overline{u}_i}{1 + \exp(\overline{u}_i \cdot \overline{v}_j)} + \sum_{i:b_{ij}=0} \frac{c_{ij}\overline{u}_i}{1 + \exp(-\overline{u}_i \cdot \overline{v}_j)} \\ &= \underbrace{-\sum_{i:b_{ij}=1} c_{ij} P(b_{ij} = 0)\overline{u}_i}_{\text{Positive Mistakes}} + \underbrace{\sum_{i:b_{ij}=0} c_{ij} P(b_{ij} = 1)\overline{u}_i}_{\text{Negative Mistakes}}\end{aligned}$$

The optimization procedure uses gradient descent to convergence:

$$\overline{u}_i \Leftarrow \overline{u}_i - \alpha \frac{\partial J}{\partial \overline{u}_i} \quad \forall i$$
$$\overline{v}_j \Leftarrow \overline{v}_j - \alpha \frac{\partial J}{\partial \overline{v}_j} \quad \forall j$$

It is noteworthy that the derivatives can be expressed in terms of the probabilities of making mistakes in predicting b_{ij}. This is common in gradient descent with log-likelihood optimization. It is also noteworthy that the derivative of the SGNS objective in Equation 3.48 yields a similar form of the gradient. The only difference is that the derivative of the SGNS objective is expressed over a smaller batch of instances, defined by a context window. We can also solve the probabilistic matrix factorization with mini-batch stochastic gradient descent.

With an appropriate choice of the mini-batch, the stochastic gradient descent of matrix factorization becomes identical to the backpropagation update of SGNS. The only difference is that SGNS samples negative entries *for each set of updates* on the fly, whereas matrix factorization fixes the negative samples up front. Of course, on-the-fly sampling can also be used with matrix factorization updates. The similarity of SGNS to matrix factorization can also be inferred by observing that the architecture of Figure 3.15(b) is almost identical to the matrix factorization architecture for recommender systems in Figure 3.12. As in the case of recommender systems, SGNS has missing (negative) entries. This is caused by the fact the negative sampling uses only a subset of the zero values. The only difference between the two cases is that the architecture of SGNS caps the output layer with sigmoid units, whereas a linear layer is used for recommender systems. However, recommender systems with implicit feedback use *logistic matrix factorization* [235], which is similar to the *word2vec* setting.

3.7 Simple Neural Architectures for Graph Embeddings

Large networks have become very common because of their ubiquity in many social- and Web-centric applications. Graphs are structural entries containing *nodes* and *edges* connecting them. For example, in a social network, each person is a node, and a friendship link between two people is an edge. A friendship link is considered *undirected* because it is symmetric between the two nodes, whereas a follower-followee link is considered directed.

Graphs are typically represented with the use of *adjacency matrices*. For a graph with n nodes, its adjacency matrix A is an $m \times n$ matrix in which the (i, j)th entry is given by the weight of the directed edge from node i to node j. In the event that the edges are not directed, the adjacency matrix symmetric.

In this particular exposition, we consider the case of very large networks like the Web, a social network, or a communication network. The goal is to embed the nodes into feature vectors, so that the graph captures the relationships between nodes. For simplicity we consider undirected graphs, although directed graphs with weights on the edges can be easily handled with very few changes to the exposition below.

Consider an $n \times n$ adjacency matrix $B = [b_{ij}]$ for a graph with n nodes. The entry b_{ij} is 1 if an undirected edge exists between nodes i and j. Furthermore, the matrix B is symmetric, because we have $b_{ij} = b_{ji}$ for an undirected graph. In order to determine the embedding, we would like to determine two $n \times p$ factor matrices U and V, so that B can be derived as a function of UV^T. In the simplest case, one can set B to exactly UV^T, which is no different than a traditional matrix factorization method for factoring graphs [4]. However, for binary matrices, one can do better and use logistic matrix factorization instead. In other words, each entry of B is generated using the matrix of Bernoulli parameters in $f(UV^T)$, where $f(\cdot)$ is the element-wise application of the sigmoid function to each entry of the matrix in its argument:

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{3.52}$$

Therefore, if $\overline{u}_i$ is the ith row of U and $\overline{v}_j$ is the jth row of V, we have the following:

$$b_{ij} \sim \text{Bernoulli distribution with parameter } f(\overline{u}_i \cdot \overline{v}_j) \tag{3.53}$$

This type of generative model is typically solved using a log-likelihood model. Furthermore, *the problem formulation is identical to the logistic matrix factorization equivalent of the SGNS model in word2vec.*

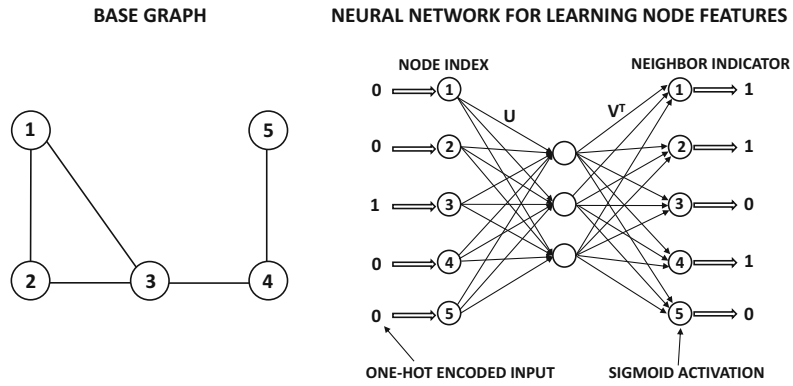


Figure 3.16: A graph of five nodes is shown together with a neural architecture for row index to neighbor indicator mapping. The shown input and output represent node 3 and its neighbors. Note the similarity to Figure 3.12. The main difference is that there are no missing values above, and the number of inputs is the same as the number of outputs for a square matrix. Both input and outputs are binary vectors. However, if negative sampling is used with sigmoid activation, most output nodes with zero values may be dropped.

Note that all *word2vec* models are logistic/multinomial variants of the model in Figure 3.12 that maps row indexes to values with linear activation. In order to explain this point, we show the neural architecture in Figure 3.16 for a toy graph containing 5 nodes. The input is the one-hot encoded index of a row in B (i.e., node), and the output is the list of all 0/1 values for all nodes in the network. In this particular case, we have shown the input for node 3 and its corresponding output. Since the node 3 has three neighbors, the output vector contains three 1s. Note that this architecture is not very different from Figure 3.12 except that it uses sigmoid activations at the output (rather than linear activations). Furthermore, since the number of 0s is usually much greater[3] than the number of 1s in the output, it is possible to drop many of the 0s with the use of negative sampling. This type of negative sampling will create a situation similar to that of Figure 3.13. With this neural architecture, the gradient-descent steps will be identical to the SGNS model of *word2vec*. The main difference is that a node appears at most once as a neighbor of another node, whereas a word might appear more than once in the context of another word. Allowing arbitrary counts on the edges takes away this distinction.

3.7.1 Handling Arbitrary Edge Counts

The aforementioned discussion assumes that the weight of each edge is binary. Consider a setting in which an arbitrary count c_{ij} is associated with the edge (i, j). In such cases, both positive and negative sampling are required. The first step is to sample an edge (i, j) from the network with probability proportional to c_{ij}. The input is, therefore, a one-hot encoded vector of the node at one end point (say, i) of this edge. The output is the one-hot encoding of node j. By default, both the input and output are n-dimensional vectors. However, if negative sampling is used, then one can reduce the output vector to a $(k + 1)$-

[3]This fact is not evident in the toy example of Figure 3.16. In practice, the degree of a node is a tiny fraction of the total number of nodes. For example, a person might have 100 friends in a social network of millions of nodes.

dimensional vector. Here, $k \ll n$ is a parameter that defines the sampling rate. A total of k negative nodes are sampled with probabilities proportional to their (weighted) degrees[4] and the outputs of these nodes are 0s. One can compute the log-likelihood loss by treating each output as the outcome of a Bernoulli trial, where the parameter of the Bernoulli trial is the output of the sigmoid activation function. The gradient descent is performed with respect to this loss. This variant is an almost exact simulation of the SGNS variant of the *word2vec* model.

3.7.2 Beyond One-Hop Structural Models

The approach discussed in the previous section can be considered a one-hop structural model, wherein the immediate adjacency of nodes can be leveraged in order create the embeddings. In practice, one would like to use more structural information in order to improve the quality of the representation. There are two ways of achieving this goal, which are discussed below.

First, one can use various types of preprocessing in order to replace the edge counts c_{ij} with multi-hop walk counts. For example, one can use random walks on the graph to (indirectly) generate c_{ij}. In this case, c_{ij} can be interpreted as the number of times that node j appears in the neighborhood of node i because it was included in a breadth-first or depth-first walk starting at node i. One can view the value of c_{ij} in the walk-based models as providing a more robust measure of the affinity between nodes i and j, as compared to the raw weights in the original graph. Of course, there is nothing sacrosanct about using a random walk to improve the robustness of c_{ij}. This broad idea forms the basis of algorithms like *node2vec* [171] and *DeepWalk* [385].

The number of choices is almost unlimited in terms of how to generate this type of affinity value. All *link prediction methods* [304] generate such affinity values between pairs of nodes, which can be used to modify edge weights. Examples include the Adamic-Adar measure, the Jaccard coefficient, and the Katz measure, all of which are discussed in [304]. The Katz measure, is also closely related to the number of random walks between a pair of nodes.

The approach of using preprocessing can be viewed as a form of hand-crafted feature engineering in order to improve the input representation. This, of course, goes against the raison d'etre of neural networks, which are intended to be end-to-end systems. Such forms of multi-hop feature engineering can be achieved with the use of *graph neural networks*, which are discussed in detail in Chapter 10. Graph neural networks construct a neural network whose hidden-layer architecture heavily depends on the structure of the base graph for which it finds an embedding. We defer a further discussion to Chapter 10.

3.7.3 Multinomial Model

The vanilla skip-gram model of *word2vec* is a multinomial model. It is also possible to use a multinomial model to create the embedding. The only difference is that the final layer of the neural network in Figure 3.16 needs to use softmax activation (instead of the sigmoid activation function). Furthermore, negative sampling is not used in the multinomial model, and both input and output layers contain exactly n nodes. As in the SGNS model, a single edge (i, j) is sampled with probability proportional to c_{ij} to create each input-output pair. The input is the one-hot encoding of i and the output is the one-hot encoding of j. One

[4]The weighted degree of node j is $\sum_r c_{rj}$.

can also use mini-batch sampling of edges to improve performance. The stochastic gradient-descent steps of this model are virtually similar to those used in the vanilla skip-gram model of *word2vec*.

3.8 Summary

This chapter discusses a number of neural models for supervised and unsupervised learning. The goal was to show that many of the traditional models used in machine learning are instantiations of relatively simple neural models. When a traditional machine learning technique like singular value decomposition is generalized to a neural representation, the results are roughly equivalent. However, the advantage of neural models is that they can usually be generalized to more powerful nonlinear models. Furthermore, it is relatively easy to experiment with nonlinear variants of traditional machine learning models with the use of neural networks. This chapter also discusses several practical applications of shallow models like recommender systems, text, and graph embeddings.

3.9 Bibliographic Notes and Software Resources

The perceptron algorithm was proposed by Rosenblatt [421], and a detailed discussion may be found in [421]. The Widrow-Hoff algorithm was proposed in [550] and is closely related to Tikhonov-Arsenin's work [516]. The Fisher discriminant was proposed by Ronald Fisher [125] in 1936, and is a specific case of the family of linear discriminant analysis methods [333]. Even though the Fisher discriminant uses an objective function that appears to be different from least-squares regression, it turns out to be a special case of least-squares regression in which the binary response variable is used as the regressand [40]. A detailed discussion of generalized linear models is provided in [331]. A variety of procedures such as *generalized iterative scaling*, *iteratively reweighted least-squares*, and *gradient descent* for multinomial logistic regression are discussed in [188]. The support-vector machine is generally credited to Cortes and Vapnik [85], although the primal method for L_2-loss SVMs was (indirectly) proposed several years earlier by Hinton [200] in the context of a neural network! This approach repairs the loss function in least-squares classification by keeping only one-half of the quadratic loss curve and setting the remaining to zero, so that it looks like a smooth version of hinge loss (try this on Figure 3.5). The specific significance of this contribution was lost within the broader literature on neural networks. The relationship of SVMs to least-squares classification is evident from related works [417], where it becomes evident that quadratic and hinge-loss SVMs are natural variations of regularized L_2-loss (i.e., Fisher discriminant) and L_1-loss classification that use the binary class variables as the regression responses [146]. The Weston-Watkins multiclass SVM was introduced in [548]. It was shown in [418] that the one-against-all approach to generalizing multiple classes seems to be as effective as the tightly integrated multiclass variants.

The earliest introduction of the autoencoder (in a more general form) is given in the backpropagation paper [424]. More detailed discussions of the autoencoder during its early years were provided in [47, 281]. A discussion of single-layer unsupervised learning may be found in [80]. Many types of regularized autoencoders (which generalize well to out-of-sample data with specific properties) are discussed in Chapter 5. The use of autoencoders for outlier detection is explored in [64, 191, 589], and a survey on the use in clustering is provided in [9]. The use of Restricted Boltzmann Machines for dimensionality reduction is discussed in [208].

An item-based autoencoder is discussed in [456], and this approach is a neural generalization of item-based neighborhood regression [261]. The main difference is that the regression weights are regularized with a constricted hidden layer. Similar works with different types of item-to-item models with the use of de-noising autoencoders are discussed in [488, 555]. A more direct generalization of matrix factorization methods may be found in [196], although the approach in [196] is slightly different from the simpler approach presented in this chapter. The incorporation of content in building recommender systems for deep learning is discussed in [533]. A multiview deep learning approach, which has also been extended to temporal recommender systems in a later work [481], is proposed in [112]. A survey of deep learning methods for recommenders may be found in [584]. The application of dimensionality reduction to recommender systems with the use of restricted Boltzmann machines may be found in [431].

The *word2vec* model was proposed in [337, 339], and a detailed exposition may be found in [420]. An alternative to the SGNS model for achieving greater efficiency is to use a hierarchical softmax layer, where each class corresponds to a leaf of a binary tree. For example, if the prediction is a target word, one can use the *WordNet* hierarchy [341] to guide the grouping. Further reorganization may be needed [355] because the *WordNet* hierarchy is not exactly a binary tree. Another option is to use Huffman encoding in order to create the binary tree [337, 339]. The basic ideas in *word2vec* have been extended to sentence- and paragraph-level embeddings, with a model, which is referred to as *doc2vec* [278]. An alternative of *word2vec* that uses a different type of matrix factorization is GloVe [384]. Multi-lingual word embeddings are presented in [10]. The extension of *word2vec* to graphs with node-level embeddings is provided in the *DeepWalk* [385] and *node2vec* [171] models. A detailed discussion of network embedding is provided in Chapter 10.

Software Resources

Machine learning models like linear regression, SVMs, and logistic regression are available from *scikit-learn* [611]. The DISSECT (Distributional Semantics Composition Toolkit) [612] is a toolkit that uses word co-occurrence counts in order to create embeddings. The GloVe method is available from Stanford NLP [613] and the **gensim** library [410]. The *word2vec* tool is available under the Apache license [615], and as a **TensorFlow** version [616]. The **gensim** library has Python implementations of *word2vec* and *doc2vec* [410]. Java versions of *doc2vec*, *word2vec*, and GloVe may be found in the **DeepLearning4j** repository [614]. In several cases, one can simply download pre-trained versions of the representations (on a large corpus that is considered generally representative of text) and use them directly, as a convenient alternative to training for the specific corpus at hand. The *node2vec* software is available from the original author at [617].

3.10 Exercises

1. Consider the following loss function for training pair $(\overline{X}, y)$:

$$L = \max\{0, a - y(\overline{W} \cdot \overline{X})\}$$

The test instances are predicted as $\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}^T\}$. A value of $a = 0$ corresponds to the perceptron criterion and a value of $a = 1$ corresponds to the SVM. Show that any value of $a > 0$ leads to the SVM with an unchanged optimal solution when no regularization is used. What happens when regularization is used?

2. Based on Exercise 1, formulate a generalized objective for the Weston-Watkins SVM.

3. Consider the unregularized perceptron update for binary classes with learning rate α. Show that using any value of α is inconsequential in the sense that it only scales up the weight vector by a factor of α. Show that these results also hold true for the multiclass case. Do the results hold true when regularization is used?

4. Show that if the Weston-Watkins SVM is applied to a data set with $k = 2$ classes, the resulting updates are equivalent to the binary SVM updates discussed in this chapter.

5. Show that if multinomial logistic regression is applied to a data set with $k = 2$ classes, the resulting updates are equivalent to logistic regression updates.

6. Implement the softmax classifier using a deep-learning library of your choice.

7. In linear-regression-based neighborhood models, the rating of an item is predicted as a weighted combination of the ratings of other items of the same user, where the item-specific weights are learned with linear regression. Show how you can construct an autoencoder architecture to create this type of model. Discuss the relationship of this architecture with the matrix factorization architecture.

8. **Logistic matrix factorization:** Consider an autoencoder which has an input layer, a single hidden layer containing the reduced representation, and an output layer with sigmoid units. The hidden layer has linear activation:

   (a) Set up a negative log-likelihood loss function for the case when the input data matrix is known to contain binary values from $\{0, 1\}$.

   (b) Set up a negative log-likelihood loss function for the case when the input data matrix contains real values from $[0, 1]$.

9. **Non-negative matrix factorization with autoencoders:** Let D be an $n \times d$ data matrix with non-negative entries. Show how you can approximately factorize $D \approx UV^T$ into two non-negative matrices U and V, respectively, by using an autoencoder architecture with d inputs and outputs. [Hint: Choose an appropriate activation function in the hidden layer, and modify the gradient-descent updates.]

10. **Probabilistic latent semantic analysis:** Refer to [216] for a definition of probabilistic latent semantic analysis. Propose a modification of the approach in Exercise 9 for probabilistic latent semantic analysis. [Hint: What is the relationship between non-negative matrix factorization and probabilistic latent semantic analysis?]

11. **Simulating a model combination ensemble:** In machine learning, a model combination ensemble averages the scores of multiple models in order to create a more robust classification score. Discuss how you can approximate the averaging of an Adaline and logistic regression with a two-layer neural network. Discuss the similarities and differences of this architecture with an actual model combination ensemble when backpropagation is used to train it. Show how to modify the training process so that the final result is a fine-tuning of the model combination ensemble.

12. **Simulating a stacking ensemble:** In machine learning, a stacking ensemble creates a higher-level classification model on top of features learned from first-level classifiers. Discuss how you can modify the architecture of Exercise 11, so that the first level of

classifiers correspond to an Adaline and a logistic regression classifier and the higher-level classifier corresponds to a support vector machine. Discuss the similarities and differences of this architecture with an actual stacking ensemble when backpropagation is used to train it. Show how you can modify the training process of the neural network so that the final result is a fine-tuning of the stacking ensemble.

13. Show that the stochastic gradient-descent updates of the perceptron, Widrow-Hoff learning, SVM, and logistic regression are all of the form $\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha y[\delta(\overline{X}, y)]\overline{X}^T$. Here, the mistake function $\delta(\overline{X}, y)$ is $1 - y(\overline{W} \cdot \overline{X}^T)$ for least-squares classification, an indicator variable for perceptron/SVMs, and a probability value for logistic regression. Assume that α is the learning rate, and $y \in \{-1, +1\}$. Write the specific forms of $\delta(\overline{X}, y)$ in each case.

14. The linear autoencoder discussed in the chapter is applied to each d-dimensional row of the $n \times d$ data set D to create a k-dimensional representation. The encoder weights contain the $k \times d$ weight matrix W and the decoder weights contain the $d \times k$ weight matrix V. Therefore, the reconstructed representation is DW^TV^T, and the aggregate loss value $\|DW^TV^T - D\|_F^2$ is minimized over the entire training data set.

(a) For a fixed value of V, show that the optimal matrix W must satisfy $D^TD(W^TV^TV - V) = 0$.

(b) Use (a) to show that if the $n \times d$ matrix D has rank d, we have $W^TV^TV = V$.

(c) Use (b) to show that $W = (V^TV)^{-1}V^T$. Assume that V^TV is invertible.

(d) Repeat exercise parts (a), (b), and (c), when the encoder-decoder weights are tied as $W = V^T$. Show that the columns of V must be orthonormal.

15. The autoencoder with shared weights in section 3.4.1.2 simulates SVD in terms of finding the same subspace in its weight matrix V as the dominant right singular vectors, but the columns of this weight matrix represent a rotation of the dominant singular vectors of SVD. Discuss how you can use iterative training of an autoencoder architecture with successive addition of hidden units to exactly determine the dominant singular vectors of SVD.

16. Maximum margin matrix factorization: The hinge loss of the SVM is designed for binary classification. Use this inspiration to design a neural architecture for matrix factorization of binary matrices with entries drawn from $\{-1, +1\}$. Design a loss function like the hinge loss for matrix factorization.

17. Multinomial matrix factorization: Suppose that you have a matrix containing categorical entries like Zip Codes, ethnic identity, and so on. Discuss how you can leverage an autoencoder architecture in conjunction with appropriate activation and loss functions in order to perform the factorization. [Hint: Look at the output layer in softmax regression.]

18. Most matrices in the real world are mixed, and may contain a combination of real-valued, binary, and categorical columns. How do your answers to Exercises 16 and 17 provide a path to dealing with such matrices?

19. Incomplete matrix with categorical entries: Suppose that you have an incomplete matrix in which each entry is a categorical value value, and the different columns of the matrix correspond to different categorical attributes (e.g., Zip Code or ethnicity). You

want to predict missing entries. Draw inspirations from the recommender architecture discussed in the chapter to propose a neural model for this case.

20. Discuss how you would modify the recommender architecture discussed in the chapter when you have a d-dimensional attribute associated with each item (in addition to an incomplete matrix of ratings). Your architecture should build on top of the one already presented in the chapter.

21. For the graph embedding algorithm discussed in the chapter, what is the difference in the training procedure for undirected and directed graphs?

22. Open-ended: Using your insights from the *word2vec* CBOW model to propose a neural network for completing a set from its subset. Each set is an unordered set of discrete (possibly repeating) elements from a finite universe.

Chapter 4

Deep Learning: Principles and Training Algorithms

> "I hated every minute of training, but I said, 'Don't quit. Suffer now and live the rest of your life as a champion.'" –Muhammad Ali

4.1 Introduction

The great power of deep learning models comes with computational challenges. One key point is that the backpropagation algorithm is rather *unstable* to minor changes in the algorithmic setting, such as the initialization point used by the approach. This instability is particularly significant when one is working with very deep networks.

Neural network parameter optimization is a *multivariable optimization problem*. These variables correspond to the weights of the connections in various layers. Multivariable optimization problems often face stability challenges because one must perform the steps along each direction in the "right" proportion. This turns out to be particularly hard in deep networks, where the this proportion is hard to control. One issue is that *a gradient only provides a rate of change over an infinitesimal horizon in each direction*, whereas an actual step has a finite length. One needs to choose steps of reasonable size in order to make any real progress in optimization. The problem is that the gradients do change over a step of finite length, and in some cases they change drastically. The complex optimization surfaces presented by deep networks are particularly treacherous in this respect, and the problem is exacerbated with poorly chosen settings (such as the initialization point or the normalization of the input features). As a result, the (easily computable) steepest-descent direction is often not the best direction to use for retaining the ability to use large steps. Small step sizes lead to slow progress, whereas the optimization surface might change in unpredictable ways with the use of large step sizes. All these issues make neural network optimization more difficult than would seem at first glance. However, many of these problems can be

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_4

avoided by carefully tailoring the gradient-descent steps to be more robust to the nature of the optimization surface. This chapter will discuss such algorithms.

Chapter Organization

This chapter is organized as follows. The next section discusses the benefits of increased depth. The common challenges of training deep networks are presented in section 4.3. Simple solutions based on neural architectures are presented in section 4.4. Gradient-descent strategies for deep learning are discussed in section 4.5. The Newton method is introduced in section 4.6, and its fast approximations are discussed in section 4.7. Batch normalization methods are introduced in section 4.8. A discussion of accelerated implementations of neural networks is found in section 4.9. The summary is presented in section 4.10.

4.2 Why Is Depth Beneficial?

As discussed in Chapter 1, the key to the benefits of depth lie in the use of nonlinear activation functions. In fact, the repeated composition of a linear function yields another linear function (cf. section 1.5); therefore, it does not increase the power of the learning algorithm. However, composing nonlinear functions repeatedly leads to inherently more powerful models. In general, each layer of the network constructs successively more complex features that are relevant to the predictive problem at hand. This principle is referred to as that of *hierarchical feature engineering*.

4.2.1 Hierarchical Feature Engineering: How Depth Reveals Rich Structure

Many deeper architectures with feed-forward architectures have multiple layers in which successive transformations of the inputs from the previous layer lead to increasingly sophisticated representations of the data. The values of each hidden layer for a particular input contain a transformed representation of the input point, which becomes increasingly informative about the target value we are trying to learn, as the layer gets closer to the output node. The nonlinear activations in intermediate layers play a critical role in the feature engineering process.

As shown in section 1.5.1 of Chapter 1, appropriately transformed feature representations are more amenable to predictive settings like classification. For example, if the final layer uses a linear separator to perform classification, the intermediate layers might transform the data to a representation where a linear separator works effectively. One way of viewing this division of labor between the hidden layers and final prediction layer is that the early layers create a feature representation that is more informative for a predictive task. The final layer then leverages this learned feature representation for prediction. This division of labor is shown in Figure 4.1. One can view this process as a kind of hierarchical feature engineering in which the features in earlier layers represent primitive characteristics of the data, whereas those in later layers represent complex characteristics with semantic significance to the class labels.

The idea behind deeper architectures is that they can better leverage *repeated regularities* in the data patterns in order to reduce the number of computational units and therefore *generalize* the learning even to areas of the data space where one does not have examples. Often these repeated regularities are learned by the neural network within the weights as the basis vectors of hierarchical features. Although a detailed proof [351] of this fact is beyond

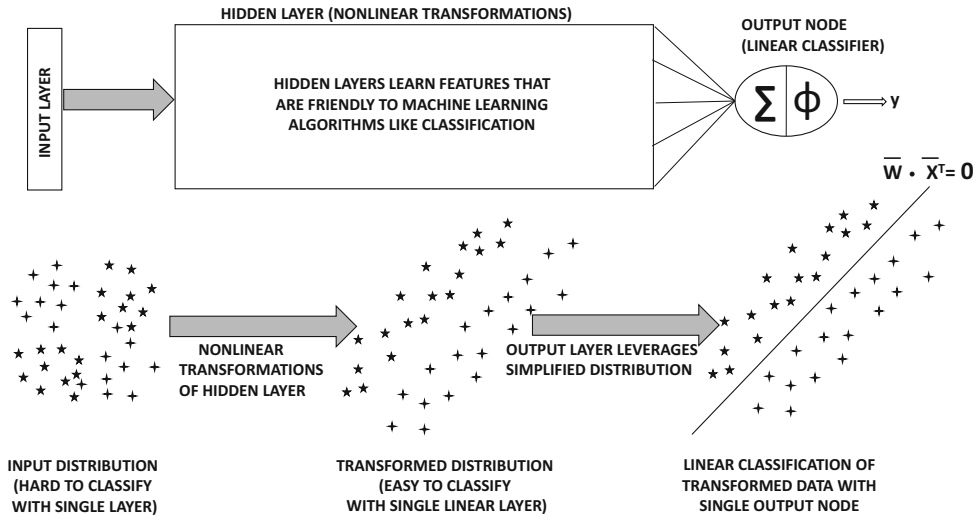


Figure 4.1: The feature engineering role of the hidden layers

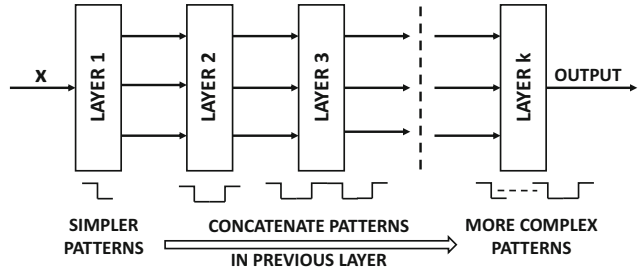


Figure 4.2: Deeper networks can learn more complex functions by composing the functions learned in earlier layers.

the scope of this book, we provide a simple example to elucidate this point. Consider a situation in which a 1-dimensional function is defined by 1024 repeated steps of the same size and height. A shallow network with one hidden layer and step activation functions would require at least 1024 units in order to model the function. However, a multilayer network would model a pattern of 1 step in the first layer, 2 steps in the next, 4 steps in the third, and 2^r steps in the rth layer. This situation is illustrated in Figure 4.2. Note that the pattern of 1 step is the simplest feature because it is repeated 1024 times, whereas a pattern of 2 steps is more complex. Therefore, the features (and the functions learned) in successive layers are hierarchically related. In this case, a total of 10 layers are required and a small number of constant nodes are required in each layer to model the joining of the two patterns from the previous layer. Therefore, the overall number of nodes required is an *order of magnitude less* than that required in the single-layer network. This means that the amount of data required for learning is also an order of magnitude less. The reason for this is that the multilayer network implicitly *looks for the repeated regularities and learns them* with a modest amount of data, rather than trying to explicitly learn every turn and twist of the target function.

One can also understand the importance of depth in terms of the fact that a multilayer neural network is a composition of functions, in which the depth of the composition corresponds to the depth of the neural network. For example if the layer i computes the vector-to-vector function $f_i(\cdot)$, then the overall function computed by a neural network with t computational layers is $f_t(f_{t-1}(\ldots(f_1(\overline{x}))))$ for input vector $\overline{x}$. We summarize this principle below:

> A repeated composition of relatively simple nonlinear functions (like neural network activation functions) can be used to create a very complex nonlinear function. The richness of the overall function computed by the neural network increases with the depth of the composition. Each layer of the neural network can learn successively more refined patterns from the patterns learned in previous layers.

This type of behavior is particularly evident in a visually interpretable way in some domains like convolutional neural networks for image data (cf. Chapter 9). In convolutional neural networks, the features in earlier layers capture detailed but primitive shapes like lines or edges from the data set of images. On the other hand, the features in later layers capture shapes of greater complexity like hexagons, honeycombs, and so forth, depending on the type of images provided as training data. Note that such semantically interpretable shapes often have closer correlations with class labels in the image domain. For example, almost any image will contain lines or edges, but images belonging to particular classes will be more likely to have hexagons or honeycombs. A later layer might model a complex shape like a face. On the other hand, a single layer would have difficulty in modeling every twist and turn of a face, because it would have a harder time in recognizing the repeated regularities of a human face, which are only possible with the perspective obtained by greater depth. This provides the deeper model with the ability to learn rich structure in the data. This property tends to make the representations of later layers easier to classify with simple models like linear classifiers. This process is illustrated in Figure 4.1. The features in earlier layers are used repeatedly as building blocks to create more complex features. This general principle of "putting together" simple features to create more complex features lies at the core of the successes achieved with neural networks.

4.3 Why Is Training Deep Networks Hard?

Increasing the depth of the network is not without its disadvantages. Deeper networks are often harder to train, and are also notoriously unstable to parameter choice. This is because depth increases the loss function complexity by recursive composition, as a result of which gradient descent behaves in unexpected ways.

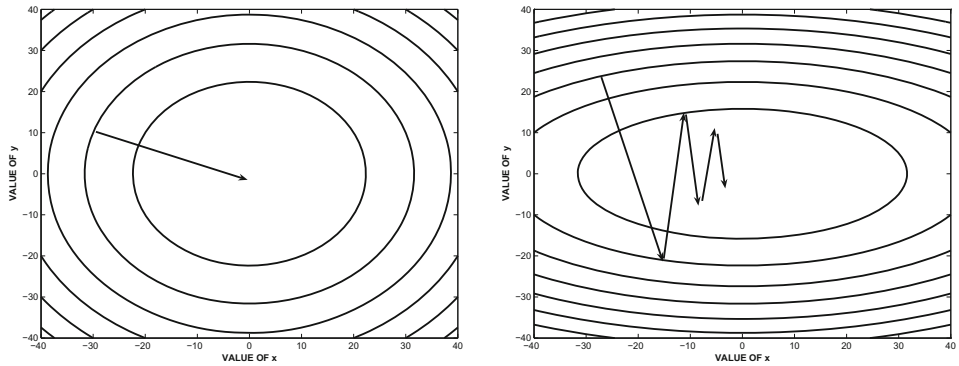
4.3.1 Geometric Understanding of the Effect of Gradient Ratios

An important problem that arises in the use of neural networks is the *vanishing and exploding gradient problem*. The basic idea underlying this problem (discussed in detail later in this chapter), is that the gradients of the loss function with respect to the weights in different layers is very different in magnitude. For example, the (magnitude of the) partial derivative with respect to a weight in the final layer might be 10^6 times the magnitude of the partial derivative with respect to a weight in the first layer. Therefore, a gradient descent update that changes a parameter in the final layer by 0.2 units will (negligibly)

change the weight in the first layer by 2×10^{-6}. In other words, the gradients have *vanished* during backpropagation. Similarly, it is possible for the gradients to explode uncontrollably during backpropagation, where the magnitudes of the gradients in early layers are much larger than the ones in later layers. This can cause problems in the optimization process in being able to make updates of meaningful size.

The vanishing and exploding gradient problems are inherent to multivariable optimization, even in cases where there are no local optima. In fact, minor manifestations of this problem are encountered in almost any convex optimization problem. Therefore, in this section, we will consider the simplest possible case of a convex, quadratic objective function with a bowl-like shape and a single global minimum. In a single-variable problem, the path of steepest descent (which is the only path of descent), will always pass through the minimum point of the bowl (i.e., optimum objective function value). However, the moment we increase the number of variables in the optimization problem from 1 to 2, this is no longer the case. The key point to understand is that *with very few exceptions, the path of steepest descent in most loss functions is only an instantaneous direction of best movement, and is not the correct direction of descent in the longer term.* In other words, small steps with "course corrections" are always needed. When an optimization problem exhibits the vanishing gradient problem, it means that the only way to reach the optimum with steepest-descent updates is by using an *extremely large number of tiny updates and course corrections*, which is obviously very inefficient.'

In order to understand this point, we look at two bivariate loss functions in Figure 4.3. In this figure, the contour plots of the loss function are shown, in which each line corresponds to points in the XY-plane where the loss function has the same value. The direction of steepest descent is always perpendicular to this line. The first loss function is of the form $L = x^2 + y^2$, which takes the shape of a perfectly circular bowl, if one were to view the height as the objective function value. This loss function treats x and y in a symmetric way. The second loss function is of the form $L = x^2 + 4y^2$, which is an elliptical bowl. Note that this loss function is more sensitive to changes in the value of y as compared to changes in the value of x, although the specific sensitivity depends on the position of the data point. In these cases, the *second-order* derivatives $\frac{\partial^2 L}{\partial x^2}$ and $\frac{\partial^2 L}{\partial y^2}$ are different in the case of the loss $L = x^2 + 4y^2$. The second-order derivatives are also referred to as the curvature, because they control how the slope changes in the two cases.



(a) Loss function is circular bowl $L = x^2 + y^2$

(b) Loss function is elliptical bowl $L = x^2 + 4y^2$

Figure 4.3: The effect of the shape of the loss function on steepest-gradient descent.

In the case of the circular bowl of Figure 4.3(a), the gradient points directly at the optimum solution, and one can reach the optimum in a single step, as long as the correct step-size is used. This is not quite the case in the loss function of Figure 4.3(b), in which the gradients are often more significant in the y-direction compared to the x-direction. Furthermore, the gradient never points to the optimal solution, as a result of which many course corrections are needed over the descent. A salient observation is that the steps along the y-direction are large, but subsequent steps undo the effect of previous steps. On the other hand, the progress along the x-direction is consistent but tiny. Although the situation of Figure 4.3(b) occurs in almost any optimization problem using steepest descent, the case of the vanishing gradient is an extreme manifestation of this behavior. The fact that a simple quadratic bowl (which is trivial compared to the typical loss function of a deep network) shows so much oscillation with the steepest-descent method is concerning. After all, the repeated composition of functions (as implied by the underlying computational graph) is highly *unstable* in terms of the sensitivity of the output to the parameters in different parts of the network. The problem of differing relative derivatives is extraordinarily large in real neural networks, in which we have millions of parameters and gradient ratios that vary by orders of magnitude. Furthermore, many activation functions have derivatives that are always less than 1, which tends to encourage the vanishing gradient problem by repeated multiplication of these derivatives (according to the chain rule) during backpropagation. As a result, the parameters in later layers with large descent components are often oscillating with large updates, whereas those in earlier layers make tiny but consistent updates. Therefore, neither the earlier nor the later layers make much progress in getting closer to the optimal solution. As a result, it is possible to get into situations where very little progress is made even after training for a long time. In the next section, we provide an understanding of this mechanism in the specific case of neural networks.

4.3.2 The Vanishing and Exploding Gradient Problems

Neural networks have millions of parameters in which some components of the gradients are extremely large as compared to others. This type of skewed gradient ratio is manifested in a particular way in neural networks, in which the partial derivatives with respect to weights in different *layers* are very different. This is primarily because of how a neural network computes a composition of functions across different layers, and how each layer of the composition incorporates a multiplicative factor to the gradient because of the mechanism of the chain rule of differential calculus. In order to understand this point, let us consider a very deep network that has a single node in each layer. We assume that there are $(m+1)$ layers, including the non-computational input layer. The weights of the edges between the various layers are denoted by $w_1, w_2, \ldots w_m$. Furthermore, assume that the sigmoid activation function $\Phi(\cdot)$ is applied in each layer. Let x be the input, $h_1 \ldots h_{m-1}$ be the hidden values in the various layers, and o be the final output. Let $\Phi'(h_t)$ be the derivative of the activation function in hidden layer t. Let $\frac{\partial L}{\partial h_t}$ be the derivative of the loss function with respect to the hidden activation h_t. The neural architecture is illustrated in Figure 4.4. Note that the output of each hidden layer is defined from the previous layer using the following relationship:

$$h_{t+1} = \Phi(w_{t+1} h_t)$$

Therefore, we have the following:

$$\frac{\partial h_{t+1}}{\partial h_t} = \Phi'(w_{t+1} h_t) w_{t+1}$$

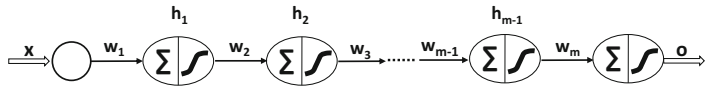


Figure 4.4: The vanishing and exploding gradient problems

One can use the chain rule in the backpropagation update to show the following relationship:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} = \Phi'(w_{t+1} h_t) \cdot w_{t+1} \cdot \frac{\partial L}{\partial h_{t+1}} \tag{4.1}$$

Since the fan-in is 1 of each node, assume that the weights are initialized from a standard normal distribution. Therefore, each w_t has an expected average magnitude of 1.

Let us examine the specific behavior of this recurrence in the case where the sigmoid activation is used. The derivative with a sigmoid with output $f \in (0, 1)$ is given by $f(1-f)$. This value takes on its maximum at $f = 0.5$, and therefore the value of $\Phi'(w_{t+1} h_t)$ is no more than 0.25 even at its maximum. Since the absolute value of w_{t+1} is expected to be 1, it follows that each weight update will (typically) cause the value of $\frac{\partial L}{\partial h_t}$ to be less than 0.25 that of $\frac{\partial L}{\partial h_{t+1}}$. Therefore, after moving by about r layers, this value will typically be less than 0.25^r. Just to get an idea of the magnitude of this drop, if we set $r = 10$, then the gradient update magnitudes drop to 10^{-6} of their original values! It is also noteworthy that even though we have shown the derivatives with respect to the hidden layers, the derivatives with respect to weights are direct multiples of those with respect to hidden layers; the trends will be similar. Therefore, when backpropagation is used, the earlier layers will receive very small updates compared to the later layers. This problem is referred to as the *vanishing gradient problem*. Note that we could try to solve this problem by using an activation function with larger gradients and also initializing the weights to be larger. However, if we go too far in doing this, it is easy to end up in the opposite situation where the gradient *explodes* in the backward direction instead of vanishing. In general, unless we initialize the weight of every edge so that the product of the weight and the derivative of each activation is exactly 1, there will be considerable instability in the magnitudes of the partial derivatives. In practice, this is impossible with most activation functions because the derivative of an activation function will vary from iteration to iteration.

Although we have used an oversimplified example here with only one node in each layer, it is easy to generalize the argument to cases in which multiple nodes are available in each layer. In general, it is possible to show that the layer-to-layer backpropagation update includes a matrix multiplication (rather than a scalar multiplication). Just as repeated scalar multiplication is inherently unstable, so is repeated matrix multiplication. In particular, the loss derivatives in layer-$(i+1)$ are multiplied by a matrix referred to as the Jacobian (cf. Equation 2.21). The Jacobian contains the derivatives of the activations in layer-$(i+1)$ with respect to those in layer i. In certain cases like recurrent neural networks, the Jacobian is a square matrix and one can actually impose stability conditions with respect to the largest eigenvalue of the Jacobian. These stability conditions are rarely satisfied exactly, and therefore the model has an inherent tendency to exhibit the vanishing and exploding gradient problems. Furthermore, the effect of activation functions like the sigmoid tends to encourage the vanishing gradient problem. One can summarize this problem as follows:

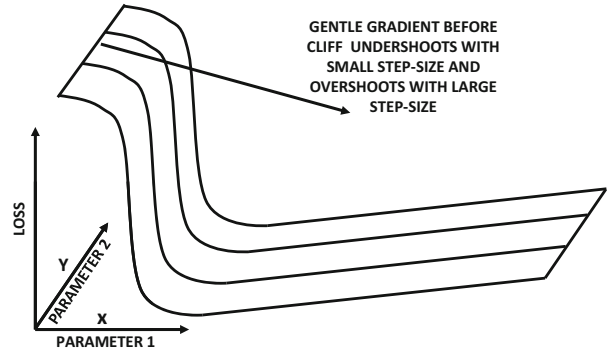


Figure 4.5: An example of a cliff in the loss surface

Observation 4.3.1 *The relative magnitudes of the partial derivatives with respect to the parameters in different parts of the network tend to be very different, which creates problems for gradient-descent methods.*

4.3.3 Cliffs and Valleys

Gradient descent works with first-order derivatives. However, the steps taken in gradient descent are of a finite size, and they change along the course of the step. The rate of change of first-order derivatives can be quantified by the second-order derivatives, which is also informally referred to as the *curvature*. A particular example of a distressing topology from the perspective of gradient descent is that of a *cliff*.

An example of a loss surface is shown in Figure 4.5. In this case, there is a gently sloping surface that rapidly changes into a steeply descending surface; this type of surface is referred to as a cliff. However, if one computed only the first-order partial derivative with respect to the variable x shown in the figure, one would only see a gentle slope. As a result, a small learning rate will lead to very slow learning, whereas increasing the learning rate can suddenly cause overshooting to a point far from the optimal solution. This problem is caused by the nature of the curvature (i.e., changing gradient), where the first-order gradient does not contain the information needed to control the size of the update. In many cases, the rate of change of gradient can be computed using the second-order derivative, which provides useful (additional) information.

Cliffs are not desirable because they manifest a certain level of instability in the loss function. This implies that a small change in some of the neural-network weights can either change the loss in a tiny way or suddenly change the loss by such a large amount that the resulting solution is even further away from the true optimum. As a result, it is easy to miss the optimum during a gradient-descent step. Such settings are often addressed with techniques that either modify the gradient in some way to favor directions of low curvature, or explicitly use the curvature (i.e., second-order derivative) of the loss function. This chapter will discuss several such algorithms.

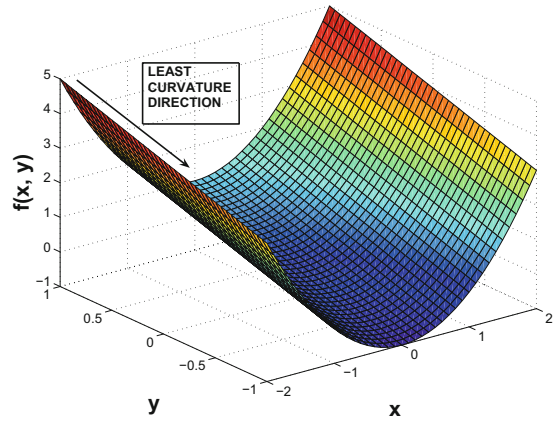


Figure 4.6: The curvature effect in valleys

The specific effect of curvature is particularly evident when one encounters loss functions in the shape of sloping or winding valleys. An example of a sloping valley is shown in Figure 4.6. A valley is a dangerous topography for a gradient-descent method, particularly if the bottom of the valley has a steep and rapidly changing surface (which creates a narrow valley). This is, of course, not the case in Figure 4.6, which is a relatively easier case. However, even in this case, the steepest-descent direction will often bounce along the sides of the valley, and move down the slope relatively slowly if the step-sizes are chosen inaccurately. In narrow valleys, the gradient-descent method will bounce along the steep sides of the valley even more violently without making much progress in the gently sloping direction, where the greatest *long-term* gains are present. It is again evident that choosing only the steepest-decent direction for steps is not helpful; rather, one must also favor low-curvature directions.

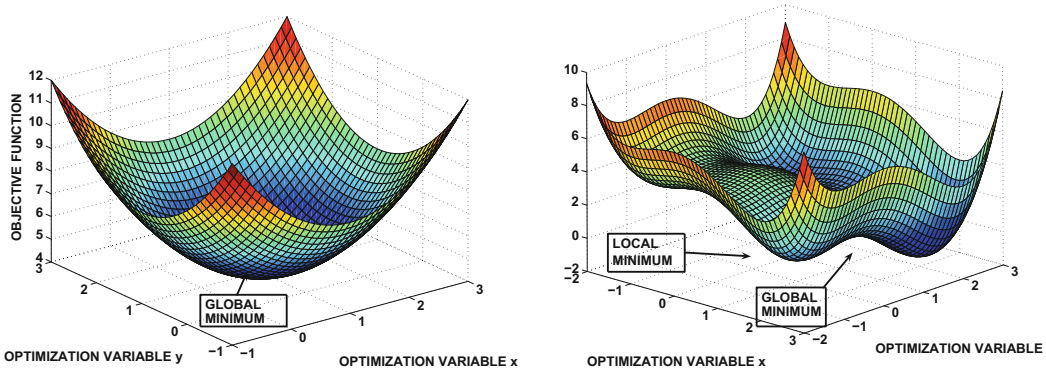
4.3.4 Convergence Problems with Depth

Very deep architectures have a difficult time in converging to an optimal solution during backpropagation. This problem has been explored in [193, 194], where it is conjectured that the rate of convergence of a deep network slows down exponentially with depth. Although the vanishing and exploding gradient problems also cause problems with convergence, the work in [193, 194] shows that using various tricks to reduce the vanishing and exploding gradient problems continue to cause problems with convergence. Therefore, the problem of convergence seems to be multi-faceted, and it cannot be easily explained by a single factor (like the vanishing and exploding gradient problems). One issue is that the partial derivatives in different layers are interdependent, and it takes a while for changes in weights in different layers to propagate to other layers (especially as depth increases). As a result, it is possible for the weights in different layers to move around without converging to an optimal solution.

4.3.5 Local Minima

Certain types of loss functions have a single global minimum. Such problems are referred to as *convex optimization problems*, and they represent the simplest case of optimization. A loss function $L(\overline{w})$ mapping to real values is said to be convex, if it satisfies the following for all weight vectors $\overline{w}_1$ and $\overline{w}_2$ and $\lambda \in (0, 1)$:

$$L(\lambda \overline{w}_1 + [1 - \lambda]\overline{w}_2) \leq \lambda L(\overline{w}_1) + (1 - \lambda)L(\overline{w}_2)$$



(a) Single global minimum
$g(x, y) = x^2 + y^2 - 2x - 2y + 6$

(b) Global and local minimum
$G(x, y) = ([x^4 + y^4]/4)$
$-([x^3 + y^3]/3) - x^2 - y^2 + 4$

Figure 4.7: Illustrations of local and global optima

A convex loss function is shaped like a bowl with a single bottom, whereas a non-convex function might have multiple "potholes" in the bowl. An example of a 2-dimensional convex loss function $g(x, y) = x^2 + y^2 - 2x - 2y + 6$ is illustrated in Figure 4.7(a), where the two horizontal axes are the variables and the vertical axis is the loss value. It is evident that this function has a single global minimum at $(x, y) = (1, 1)$. On the other hand, the loss function $G(x, y)$ in Figure 4.7(b) is not convex and it has multiple local minima.

The convexity of a function is important from the perspective of its behavior during gradient descent. When gradient descent reaches such a pothole, it might be unable to escape from that point because the gradient at the bottom of a pothole is 0 and all instantaneous directions of movement worsen the loss function. Most of the shallow neural networks discussed in Chapter 3 have convex loss functions, and therefore such a situation does not arise. A proof of convexity of many of these loss functions may be found in [6]. The general reason is that most of these shallow architectures can be explained by a single composition $g(f(\cdot))$ of a convex nonlinear loss function $g(\cdot)$ and a linear function $f(\cdot)$. Such a composition of functions maintains convexity of the loss function [6] as long as the convex nonlinear function is applied as the outer nest of this composition. For example, the linear function $f(x_1, x_2) = x_1 - x_2$ and quadratic function $g(x) = x^2$ are both convex, and the function $g(f(x_1, x_2)) = (x_1 - x_2)^2$ is convex. However, with increasing depth, the loss function is often not convex. This is primarily because most activation functions are convex, although composing a linear function and nonlinear convex function *multiple* times (or in the "wrong" order) is not always convex. For example, the linear function $f(x_1, x_2) = x_1 - x_2$ and quadratic function $g(x) = x^2$ are both convex, but the function $f(g(x_1), g(x_2)) = x_1^2 - x_2^2$ is not convex.

A deep neural network is the result of the composition of many linear dot products and nonlinear activation functions. In such cases, the overall function computed by the network is no longer convex in either the inputs or the weight variables. Therefore, gradient-descent is no longer guaranteed to converge to a global optimum. The procedure could easily get stuck in one of the "potholes" of the loss function; these spurious "minima" are also referred to as *local optima*.

Local minima are problematic only when their objective function values are significantly larger than that of the global minimum. In practice, however, this does not seem to be the case in neural networks. Many research results [91, 443] have shown that the local minima of real-life networks have very similar objective function values to the global minimum. As a result, their presence does not seem to cause a serious problem. Furthermore, some of the training methods discussed in this chapter, such as *momentum methods*, are very well suited to avoiding local optima during gradient descent.

4.4 Depth-Friendly Neural Architectures

Certain types of neural architectures are easier to training than others. The specific choice of the activation function and the way in which the units are connected both have an effect on gradient descent. This section will discuss various depth-friendly architectural tricks.

4.4.1 Activation Function Choice

The specific choice of activation function often has a considerable effect on the severity of the vanishing gradient problem. The derivatives of the sigmoid and the tanh activation functions are illustrated in Figure 4.8(a) and (b), respectively. The sigmoid activation function never has a gradient of more than 0.25, and therefore it is very prone to the vanishing gradient problem. Furthermore, it *saturates* at large absolute values of the argument, which refers to the fact that the gradient is almost 0. In such cases, the weights of the neuron change very slowly. Therefore, a few such activations within the network can significantly affect the gradient computations. The tanh function fares better than the sigmoid function because it has a gradient of 1 near the origin, but the gradient saturates rapidly at increasingly large

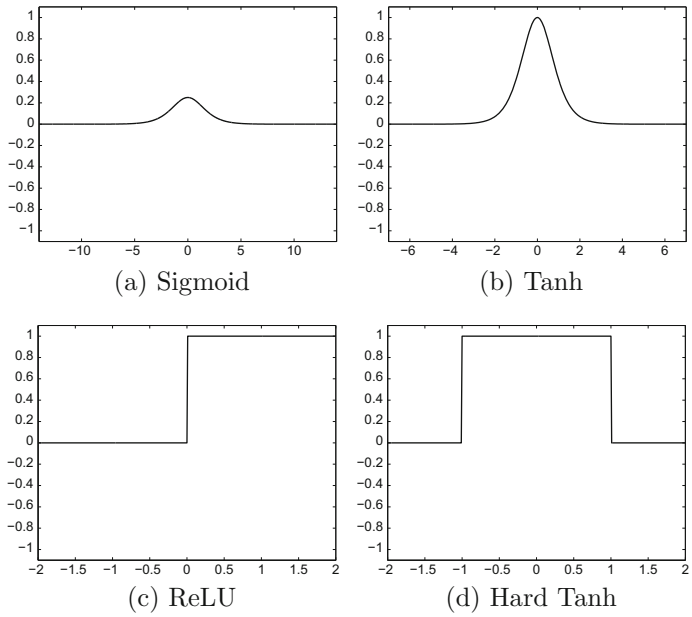


Figure 4.8: The derivatives of different activation functions

absolute values of the argument. Therefore, the tanh function will also be susceptible to the vanishing gradient problem.

In recent years, the use of the sigmoid and the tanh activation functions has been increasingly replaced with the ReLU and the hard tanh functions. The ReLU is faster to train because its gradient computation amounts to checking nonnegativity. The derivatives of the ReLU and the hard tanh functions are shown in Figures 4.8(c) and (d), respectively. It is evident that these functions take on the derivative of 1 in large intervals, although the derivative is 0 outside them. As a result, the vanishing gradient problem tends to occur less often, and these piecewise linear variants have become increasingly popular compared to their smooth counterparts. However, the use of the ReLU is only a partial fix because it does not account for the gradient flows associated with matrix multiplication across layers. Furthermore, the piecewise linear activations introduce a new problem of *dead neurons*.

4.4.2 Dying Neurons and "Brain Damage"

It is evident from Figure 4.8(c) that the derivative of the ReLU is zero for negative arguments. This can occur for a variety of reasons. For example, consider the case where the input into a neuron is always nonnegative, whereas all the weights have somehow been initialized to negative values. Therefore, the output will be 0. Another example is the case where a high learning rate is used. In such a case, the pre-activation values of the ReLU can jump to a range where the gradient is 0 irrespective of the input. In other words, high learning rates can "knock out" ReLU units. In such cases, the ReLU might not fire for any data instance. Once a neuron reaches this point, the gradient of the loss with respect to the weights just before the ReLU will always be zero. In other words, the weights of this neuron will never be updated further during training. Furthermore, its output will not vary across different choices of inputs and therefore will not play a role in discriminating between different instances. Such a neuron can be considered *dead*, which is considered a kind of permanent "brain damage" in biological parlance. The problem of dying neurons can be partially ameliorated by using learning rates that are somewhat modest. Another fix is to use the *leaky ReLU*.

4.4.2.1 Leaky ReLU

The leaky ReLU is defined using an additional parameter $\alpha \in (0, 1)$:

$$\Phi(v) = \begin{cases} \alpha \cdot v & v \leq 0 \\ v & \text{otherwise} \end{cases} \tag{4.2}$$

Therefore, at negative values of v, the leaky ReLU can still propagate (i.e., *leak*) some gradient backwards, albeit at a reduced rate defined by $\alpha < 1$. Although α is a hyperparameter chosen by the user, it is also possible to learn it.

The gains with the leaky ReLU are not guaranteed. A key point is that dead neurons are not always a problem, because they represent a kind of pruning that defines the neural network structure. Therefore, a certain level of dropping of neurons can be viewed as a part of the learning process. After all, there are limitations to our ability to tune the number of neurons in each layer. Dying neurons do a part of this tuning for us. Indeed, the intentional pruning of *connections* is sometimes used as a strategy for regularization [288]. Of course, if a very large fraction of the neurons in the network are dead, that can be a problem as

well because much of the neural network will be inactive. Furthermore, it is undesirable for too many neurons to be knocked out during the early training phases, when the model is very poor.

4.4.2.2 Maxout Networks

A recently proposed solution is the use of *maxout units* [155], which leverage two coefficient vectors $\overline{W_1}$ and $\overline{W_2}$ instead of a single one. The maxout unit computes the function $\max\{\overline{W_1} \cdot \overline{X}, \overline{W_2} \cdot \overline{X}\}$. In the event that bias neurons are used, the maxout activation is $\max\{\overline{W_1} \cdot \overline{X} + b_1, \overline{W_2} \cdot \overline{X} + b_2\}$. One can view the maxout as a generalization of the ReLU, because the ReLU is obtained by setting one of the coefficient vectors to 0. Even the leaky ReLU can be shown to be a special case of maxout, in which we set $\overline{W_2} = \alpha \overline{W_1}$ for $\alpha \in (0, 1)$. Like the ReLU, the maxout function is piecewise linear. However, it does not saturate at all, and is linear almost everywhere. In spite of its linearity, it has been shown [155] that maxout networks are universal function approximators. Maxout has advantages over the ReLU, and it enhances the performance of ensemble methods like *Dropout* (cf. section 5.5.4 of Chapter 5). However, one drawback with maxout is that it doubles the number of parameters.

4.4.3 Using Skip Connections

Most neural networks are arranged in layer-wise fashion, where connections from layer i always flow to layer $(i + 1)$. A recent idea for improving the convergence behavior of a neural network is the use of *skip connections*, wherein nodes in layer i allowed to connect to nodes in layer $(i + k)$ for $k > 1$. This idea is used extensively in image recognition, and a well-known network, referred to as *ResNet* [193, 194], has been designed with more than 100 layers. An example of a skip connection between layer i and layer $(i+2)$ is illustrated in Figure 4.9. The basic idea behind skip connections is that the network uses as much depth as it needs to in order to learn features with varying levels of richness. Simple features are learned by using skip connections frequently, whereas intricate features are learned using a large number of layers. At the same time, the network is easier to train because most of basic features can be trained using a small number of layers. This basic idea is referred to as *residual learning*, because greater depth simply learns the "residual" patterns in the data that cannot be learned using a smaller number of layers. The general principle is to *use additional layers to learn more refined features but not forget the useful features learned in earlier layers.*

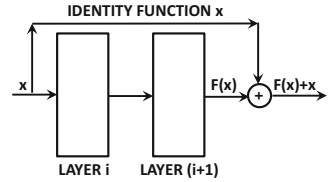


Figure 4.9: Example of a skip connection between alternate layers

4.5 Depth-Friendly Gradient-Descent Strategies

The steepest-descent direction is the optimal direction only from the perspective of infinitesimal steps, and it can worsen the objective function over a larger step. As a result, many course corrections are needed. A specific example of this phenomenon is discussed in section 4.3.1 in which minor differences in sensitivity to different features can cause a steepest-descent algorithm to have oscillations. The problem of oscillation and zigzagging is severe whenever the step is executed along a direction of *high curvature* in the loss function. This section discuss several clever learning strategies that work well in these settings.

4.5.1 Importance of Preprocessing and Initialization

The preprocessing and initialization strategies discussed in section 2.7 of Chapter 2 are extremely important for avoiding ill conditioning during gradient descent. In general, the gradients are stable across layers when the magnitudes of the activations across different layers are stable as well. Stability can be achieved by preprocessing and normalization as follows:

- *Importance of feature normalization:* When the variances of some features are much larger than others, it causes some weights to have much larger influence on the loss function than others. The varying sensitivity of the loss function to different weights can cause the type of zigzagging behavior discussed in section 4.3. Therefore, a common approach is to use standardization in order to make the variance of all inputs equal. Whitening also decorrelates the features and consolidates correlated moves of the weights into fewer non-redundant weights. Feature centering also helps because it causes different input feature values to have different signs — this ensures that all weights pointing into the same neuron from the input layer are not constrained to move in the same direction (which avoids zigzagging).

- *Importance of initialization:* A poor initialization in a deep network can exacerbate the vanishing and exploding gradient problems. With certain types of initialized weights, the vanishing and exploding gradient problems can be exacerbated in the very beginning, from which it becomes hard for gradient descent to recover. The initialized weights of the connections to a node are set to be proportional to the inverse of the square-root of the fan-in of that node by sampling them from a normal distribution with zero mean and standard deviation parameter set to $1/\sqrt{r_{in}}$. The idea is that each weight contributes a variance proportional to $1/r_{in}$, and the additive effect over nodes with different values of the fan-in becomes similar. However, this type of initialization does not account for the fact that the fan-out of the nodes also has an effect on the magnitudes of the activations in later layers. Let r_{out} be the fan-out for a particular neuron. The *Xavier initialization* or *Glorot initialization* uses a normal distribution with zero mean and standard deviation of $\sqrt{2/(r_{in} + r_{out})}$.

Normalization of features is helpful not only in the initial layer, but also in later layers. As discussed in section 4.8, more benefits may be obtained by normalization of the activations of hidden layers by using an idea called *batch normalization*.

4.5.2 Momentum-Based Learning

Momentum-based methods address the issues of local optima, flat regions, and curvature-centric zigzagging by recognizing that *emphasizing medium-term to long-term directions of consistent movement* is beneficial, because they de-emphasize local distortions in the loss topology. Consequently, an aggregated measure of the feedback from previous steps is used in order to speed up the gradient-descent procedure. As an analogy, a marble that rolls down a sloped surface with many potholes and other distortions is often able to use its momentum to overcome such minor obstacles.

Consider a setting in which one is performing gradient-descent with respect to the parameter vector $\overline{W}$. The normal updates for gradient-descent with respect to the loss function L (defined over a mini-batch of instances) are as follows:

$$\overline{V} \Leftarrow -\alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Here, α is the learning rate. We are using the matrix-calculus convention[1] that the derivative of a scalar L with respect to a column vector $\overline{W}$ is the column vector ∇L:

$$\nabla L = \frac{\partial L}{\partial \overline{W}} = \left[\frac{\partial L}{\partial w_1} \cdots \frac{\partial L}{\partial w_d} \right]^T$$

In momentum-based descent, the vector $\overline{V}$ inherits a fraction β of its velocity from its previous step in addition to the current gradient, where $\beta \in (0, 1)$ is the momentum parameter:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Setting $\beta = 0$ specializes to straightforward mini-batch gradient-descent. Larger values of $\beta \in (0, 1)$ help the approach pick up a consistent velocity $\overline{V}$ in the correct direction. The momentum parameter β is also referred to as the *friction parameter*. The word "friction" is derived from the fact that small values of β act as "brakes," much like friction. Setting $\beta > 1$ can cause instability, where the gradient descent might move in an uncontrolled way towards increasingly large loss values.

Momentum helps the gradient descent process in navigating flat regions and local optima. A good analogy for momentum-based methods is to visualize them in a similar way as a marble rolls down a bowl. As the marble picks up speed, it will be able to navigate flat regions of the surface quickly and escape form local potholes in the bowl. This is because the gathered momentum helps it escape potholes. Figure 4.10, which shows a marble rolling down a complex loss surface (picking up speed as it rolls down), illustrates this concept. The use of momentum will often cause the solution to slightly overshoot in the direction where velocity is picked up, just as a marble will overshoot when it is allowed to roll down a bowl. However, with the appropriate choice of β, it will still perform better than a situation in which momentum is not used. The momentum-based method will generally perform better because the marble gains speed as it rolls down the bowl; the quicker arrival at the optimal solution more than compensates for the overshooting of the target. Overshooting is desirable to the extent that it helps avoid local optima. The parameter β controls the amount of friction that the marble encounters while rolling down the loss surface. While increased values of β help in avoiding local optima, it might also increase oscillation at the end. In this sense, the momentum-based method has a neat interpretation in terms of the physics of a marble rolling down a complex loss surface.

[1] This convention is referred to as the *denominator layout* of matrix calculus [6]. In the numerator layout, the derivative of a scalar with respect to a column vector is a row vector.

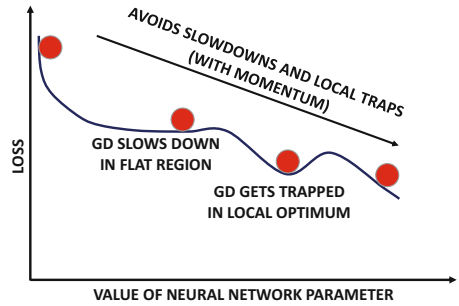


Figure 4.10: Effect of momentum in navigating complex loss surfaces.

In addition, momentum-based methods help in reducing the undesirable effects of curvature in the loss surface of the objective function. Momentum-based techniques recognize that zigzagging is a result of highly contradictory steps that cancel out one another and reduce the *effective* size of the steps in the correct (long-term) direction. An example of this scenario is illustrated in Figure 4.3(b). Simply attempting to increase the size of the step in order to obtain greater movement in the correct direction might actually move the current solution even further away from the optimum solution. In this point of view, it makes a lot more sense to move in an "averaged" direction of the last few steps, so that the zigzagging is smoothed out. This type of averaging is achieved by using the momentum from the previous steps. Oscillating directions do not contribute consistent velocity to the update.

With momentum-based descent, one is generally moving in a direction that often points closer to the optimal solution and the useless "sideways" oscillations are muted. The basic idea is to give greater preference to *consistent* directions over multiple steps, which have greater importance in the descent. This allows the use of larger steps in the correct direction without causing deteriorations in the sideways direction. As a result, learning is accelerated. An example of the use of momentum is illustrated in Figure 4.11. It is evident from Figure 4.11(a) that momentum increases the relative component of the gradient in the correct direction. The corresponding effects on the updates are illustrated in Figures 4.11(b) and (c). It is evident that momentum-based updates can reach the optimal solution in fewer updates. One can also understand this concept by visualizing the movement of a marble down the valley of Figure 4.6. As the marble gains speed down the gently sloping valley, the effects of bouncing along the sides of the valley will be muted over time.

4.5.3 Nesterov Momentum

The Nesterov momentum [364] is a modification of the traditional momentum method in which the gradients are computed at a point that would be reached after executing a β-discounted version of the previous step again (i.e., the momentum portion of the current step). This point is obtained by multiplying the previous update vector $\overline{V}$ with the friction parameter β and then computing the gradient at $\overline{W} + \beta\overline{V}$. The idea is that this corrected gradient uses a better understanding of how the gradients will change because of the momentum portion of the update, and incorporates this information into the gradient portion of the update. Therefore, one is using a certain amount of lookahead in computing the updates. Let us denote the loss function by $L(\overline{W})$ at the current solution $\overline{W}$. In this case, it is important to explicitly denote the argument of the loss function because of the way in which the gradient is computed. Therefore, the update may be computed as follows:

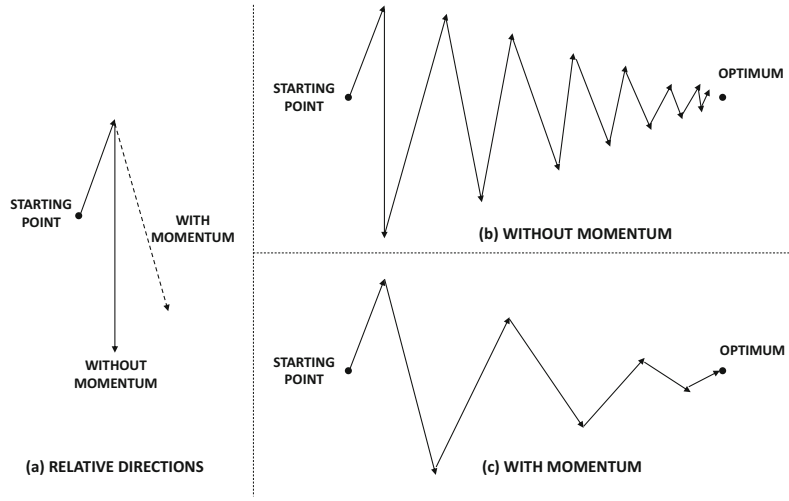


Figure 4.11: Effect of momentum in smoothing zigzagging updates

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L(\overline{W} + \beta \overline{V})}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Note that the *only* difference from the standard momentum method is in terms of *where* the gradient is computed. Using the value of the gradient a little further along the previous update can lead to faster convergence. In the previous analogy of the rolling marble, such an approach will start applying the "brakes" on the gradient-descent procedure when the marble starts reaching near the bottom of the bowl, because the lookahead will "warn" it about the reversal in gradient direction.

The Nesterov method works only in mini-batch gradient descent with modest batch sizes; using very small batches is a bad idea. In such cases, it can be shown that the Nesterov method reduces the error to $O(1/t^2)$ after t steps, as compared to an error of $O(1/t)$ in the momentum method.

4.5.4 Parameter-Specific Learning Rates

The basic idea in the momentum methods of the previous section is to leverage the *consistency* in the gradient direction of certain parameters in order to speed up the updates. This goal can also be achieved more explicitly by having different learning rates for different parameters. The idea is that parameters with large partial derivatives are often oscillating and zigzagging, whereas parameters with small partial derivatives tend to be more consistent but move in the same direction. An early method, which was proposed in this direction, was the *delta-bar-delta* method [228]. This approach tracks whether the sign of each partial derivative changes or stays the same. If the sign of a partial derivative stays consistent, then it is indicative of the fact that the direction is correct. In such a case, the partial derivative in that direction increases. On the other hand, if the sign of the partial derivative flips all the time, then the partial derivative decreases. However, this kind of approach is designed for gradient descent rather than stochastic gradient descent, because the errors in stochastic gradient descent can get magnified. Therefore, a number of methods have been proposed that can work well even when the mini-batch method is used.

4.5.4.1 AdaGrad

In the AdaGrad algorithm [110], one keeps track of the aggregated squared magnitude of the partial derivative with respect to each parameter over the course of the algorithm. The square-root of this value is *proportional* to the root-mean-square slope for that parameter (although the absolute value will increase with the number of epochs because of successive aggregation).

Let A_i be the aggregate value for the ith parameter. Therefore, in each iteration, the following update is performed:

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2 ; \quad \forall i \tag{4.3}$$

The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i}\right) ; \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}.

Scaling the derivative inversely with $\sqrt{A_i}$ is a kind of "signal-to-noise" normalization because A_i only measures the historical magnitude of the gradient rather than its sign; it encourages faster *relative* movements along gently sloping directions with consistent sign of the gradient. If the gradient component along the ith direction keeps wildly fluctuating between $+100$ and -100, this type of magnitude-centric normalization will penalize that component far more than another gradient component that consistently takes on the value in the vicinity of 0.1 (but with a consistent sign). For example, in Figure 4.11, the movements along the oscillating direction will be de-emphasized, and the movement along the consistent direction will be emphasized. However, absolute movements along all components will tend to slow down over time, which is the main problem with the approach. The slowing down is caused by the fact that A_i is the *aggregate* value of the entire history of partial derivatives. This will lead to diminishing values of the scaled derivative. As a result, the progress of AdaGrad might prematurely become too slow, and it will eventually (almost) stop making progress. Another problem is that the aggregate scaling factors depend on ancient history, which can eventually become stale. The use of stale scaling factors can increase inaccuracy. As we will see later, most of the other methods use exponential averaging, which solves both problems.

4.5.4.2 RMSProp

The RMSProp algorithm [204] uses a similar motivation as AdaGrad for performing the "signal-to-noise" normalization with the absolute magnitude $\sqrt{A_i}$ of the gradients. However, instead of simply adding the squared gradients to estimate A_i, it uses *exponential averaging*. Since one uses *averaging* to normalize rather than *aggregated* values, the progress is not slowed prematurely by a constantly increasing scaling factor A_i. The basic idea is to use a decay factor $\rho \in (0, 1)$, and weight the squared partial derivatives occurring t updates ago by ρ^t. Note that this can be easily achieved by multiplying the current squared aggregate (i.e., *running* estimate) by ρ and then adding $(1 - \rho)$ times the current (squared) partial derivative. The running estimate is initialized to 0. This causes some (undesirable) bias in

early iterations, which disappears over the longer term. Therefore, if A_i is the exponentially averaged value of the ith parameter w_i, we have the following way of updating A_i:

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2; \quad \forall i \tag{4.4}$$

The square-root of this value for each parameter is used to normalize its gradient. Then, the following update is used for (global) learning rate α:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}. Another advantage of RMSProp over AdaGrad is that the importance of ancient (i.e., stale) gradients decays exponentially with time. Furthermore, it can benefit by incorporating concepts of momentum within the computational algorithm (cf. sections 4.5.5.1 and 4.5.5.2). The drawback of RMSProp is that the running estimate A_i of the second-order moment is biased in early iterations because it is initialized to 0.

4.5.4.3 AdaDelta

The AdaDelta algorithm [578] uses a similar update as RMSProp, except that it eliminates the need for a global learning parameter by computing it as a function of incremental updates in previous iterations. Consider the update of RMSProp, which is repeated below:

$$w_i \Leftarrow w_i - \underbrace{\frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

We will show how α is replaced with a value that depends on the previous incremental updates. In each update, the value of Δw_i is the increment in the value of w_i. As with the exponentially smoothed gradients A_i, we keep an exponentially smoothed value δ_i of the values of Δw_i in previous iterations with the same decay parameter ρ:

$$\delta_i \Leftarrow \rho \delta_i + (1-\rho)(\Delta w_i)^2; \quad \forall i \tag{4.5}$$

For a given iteration, the value of δ_i can be computed using only the iterations before it because the value of Δw_i is not yet available. On the other hand, A_i can be computed using the partial derivative in the current iteration as well. This is a subtle difference between how A_i and δ_i are computed. This results in the following AdaDelta update:

$$w_i \Leftarrow w_i - \underbrace{\sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

It is noteworthy that a parameter α for the learning rate is completely missing from this update. The AdaDelta method shares some similarities with second-order methods because the ratio $\sqrt{\frac{\delta_i}{A_i}}$ in the update is a heuristic surrogate for the inverse of the second derivative of the loss with respect to w_i [578]. As discussed in subsequent sections, many second-order methods like the Newton method also do not use learning rates.

4.5.5 Combining Parameter-Specific Learning and Momentum

Although parameter-specific learning can help in differentiating between the consistency of movement along different directions, they do not gain speed along a particular direction because of consistent movement (like momentum methods). Several methods combine parameter-specific learning with momentum methods in order to gain advantages from both.

4.5.5.1 RMSProp with Nesterov Momentum

RMSProp can be combined with Nesterov momentum. Let A_i be the squared aggregate of the ith weight. In such cases, we introduce the additional parameter $\beta \in (0, 1)$ and use the following updates:

$$v_i \Leftarrow \beta v_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right); \quad w_i \Leftarrow w_i + v_i; \quad \forall i$$

Note that the partial derivative of the loss function is computed at a shifted point, as is common in the Nesterov method. The weight $\overline{W}$ is shifted with $\beta\overline{V}$ while computing the partial derivative with respect to the loss function. The maintenance of A_i is done using the shifted gradients as well:

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right)^2; \quad \forall i \tag{4.6}$$

Although this approach benefits from adding momentum to RMSProp, it does not correct for the initialization bias.

4.5.5.2 Adam

The Adam algorithm uses a similar "signal-to-noise" normalization as AdaGrad and RMSProp; however, it also incorporates momentum into the update. In addition, it directly addresses the initialization bias inherent in the exponential smoothing of pure RMSProp.

As in the case of RMSProp, let A_i be the exponentially averaged value of the ith parameter w_i. This value is updated in the same way as RMSProp with the decay parameter $\rho \in (0, 1)$:

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L}{\partial w_i} \right)^2; \quad \forall i \tag{4.7}$$

At the same time, an exponentially smoothed value of the gradient is maintained for which the ith component is denoted by F_i. This smoothing is performed with a different decay parameter ρ_f:

$$F_i \Leftarrow \rho_f F_i + (1 - \rho_f) \left(\frac{\partial L}{\partial w_i} \right); \quad \forall i \tag{4.8}$$

This type of exponentially smoothing of the gradient with ρ_f is a variation of the momentum method discussed in section 4.5.2 (which is parameterized by a friction parameter β instead of ρ_f). Then, the following update is used at learning rate α_t in the tth iteration:

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}} F_i; \quad \forall i$$

There are two key differences from the RMSProp algorithm. First, the gradient is replaced with its exponentially smoothed value in order to incorporate momentum. Second, the learning rate α_t now depends on the iteration index t, and is defined as follows:

$$\alpha_t = \alpha \underbrace{\left(\frac{\sqrt{1-\rho^t}}{1-\rho_f^t} \right)}_{\text{Adjust Bias}} \tag{4.9}$$

Technically, the adjustment to the learning rate is actually a bias correction factor that is applied to account for the unrealistic initialization of the two exponential smoothing mechanisms, and it is particularly important in early iterations. Both F_i and A_i are initialized to 0, which causes bias in early iterations. The two quantities are affected differently by the bias, which accounts for the ratio in Equation 4.9. It is noteworthy that each of ρ^t and ρ_f^t converge to 0 for large t because $\rho, \rho_f \in (0, 1)$. As a result, the initialization bias correction factor of Equation 4.9 converges to 1, and α_t converges to α. The default suggested values of ρ_f and ρ are 0.9 and 0.999, respectively, according to the original Adam paper [251]. Refer to [251] for details of other criteria (such as parameter sparsity) used for selecting ρ and ρ_f. Like other methods, Adam uses $\sqrt{A_i + \epsilon}$ (instead of $\sqrt{A_i}$) in the denominator of the update for better conditioning. The Adam algorithm is extremely popular because it incorporates most of the advantages of other algorithms, and often performs competitively with respect to the best of the other methods [251].

4.5.6 Gradient Clipping

Gradient clipping is a technique that is used to deal with settings in which the partial derivatives along different directions have exceedingly different magnitudes. Some forms of gradient clipping make the different components of the partial derivatives more similar to one another. Two forms of gradient clipping are most common:

1. *Value-based clipping:* In value-based clipping, a minimum and maximum threshold are set on the gradient values. All partial derivatives that are less than the minimum are set to the minimum threshold. All partial derivatives that are greater than the maximum are set to the maximum threshold.

2. *Norm-based clipping:* In this case, the entire gradient vector is normalized by the L_2-norm of the entire vector. Note that this type of clipping does not change the relative magnitudes of the updates along different directions. However, for neural networks that share parameters across different layers (like *recurrent neural networks*), the effect of the two types of clipping is very similar. By clipping, one can achieve a better conditioning of the values, so that the updates from mini-batch to mini-batch are roughly similar. Therefore, it would prevent an anomalous gradient explosion in a particular mini-batch from affecting the solution too much.

By and large, the effects of gradient clipping are quite limited compared to many other methods. However, it is particularly effective in avoiding the exploding gradient problem in recurrent neural networks (cf. Chapter 8).

4.5.7 Polyak Averaging

One of the motivations for second-order methods is to avoid the kind of bouncing behavior caused by high-curvature regions. The example of the bouncing behavior caused in valleys

(cf. Figure 4.6) is another example of this setting. One way of achieving some stability with any learning algorithm is to create an exponentially decaying average of the parameters over time, so that the bouncing behavior is avoided. Let $\overline{W}_1 \ldots \overline{W}_T$, be the sequence of parameters found by any learning method over the full sequence of T steps. In the simplest version of Polyak averaging, one simply computes the average of all the parameters as the final set $\overline{W}_T^f$:

$$\overline{W}_T^f = \frac{\sum_{i=1}^T \overline{W}_i}{T} \tag{4.10}$$

For simple averaging, we only need to compute $\overline{W}_T^f$ once at the end of the process, and we do not need to compute the values at $1 \ldots T-1$.

However, for exponential averaging with decay parameter $\beta < 1$, it is helpful to compute these values iteratively and maintain a running average over the course of the algorithm:

$$\overline{W}_t^f = \frac{\sum_{i=1}^t \beta^{t-i} \overline{W}_i}{\sum_{i=1}^t \beta^{t-i}} \qquad \text{[Explicit Formula]}$$

$$\overline{W}_t^f = (1-\beta)\overline{W}_t + \beta \overline{W}_{t-1}^f \quad \text{[Recursive Formula]}$$

The two formulas above are approximately equivalent at large values of t. The second formula is convenient because it enables maintenance over the course of the algorithm, and one does not need to maintain the entire history of parameters. Exponentially decaying averages are more useful than simple averages to avoid the effect of stale points. In simple averaging, the final result may be too heavily influenced by the early points, which are poor approximations to the correct solution.

4.6 Second-Order Derivatives: The Newton Method

A number of methods have been proposed in recent years for using second-order derivatives for optimization. Such methods can partially alleviate some of the problems caused by the high curvature of the loss function.

Consider the parameter vector $\overline{W} = [w_1 \ldots w_d]^T$, which is expressed as a column vector. The second-order derivatives of the loss function $L(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 L(\overline{W})}{\partial w_i \partial w_j}$$

Note that the partial derivatives use all pairwise parameters in the denominator. Therefore, for a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i,j)th entry is H_{ij}.

One can write a quadratic approximation of the loss function in the vicinity of parameter vector $\overline{W}_0$ by using the following Taylor expansion:

$$L(\overline{W}) \approx L(\overline{W}_0) + (\overline{W} - \overline{W}_0)^T [\nabla L(\overline{W}_0)] + \frac{1}{2}(\overline{W} - \overline{W}_0)^T H (\overline{W} - \overline{W}_0) \tag{4.11}$$

The accuracy of this approximation falls off with increasing value of $||\overline{W} - \overline{W}_0||$. which is the Euclidean distance between $\overline{W}$ and $\overline{W}_0$. Note that the Hessian H is computed at $\overline{W}_0$. Here, the parameter vectors $\overline{W}$ and $\overline{W}_0$ are d-dimensional column vectors, as is the gradient

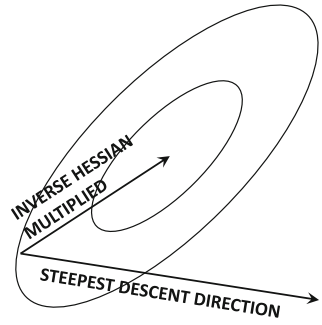


Figure 4.12: The effect of pre-multiplication of steepest-descent direction with the inverse Hessian

of the loss function. This is a quadratic approximation, and one can simply set the gradient to 0, which results in the following optimality condition for the quadratic approximation:

$$\nabla L(\overline{W}) = 0 \quad \text{[Gradient of Loss Function]}$$
$$\nabla L(\overline{W}_0) + H(\overline{W} - \overline{W}_0) = 0 \quad \text{[Gradient of Taylor approximation]}$$

One can rearrange the above optimality condition to obtain the following Newton update:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla L(\overline{W}_0)] \tag{4.12}$$

The main difference of Equation 4.12 from the update of steepest-gradient descent is pre-multiplication of the steepest direction (which is $[\nabla L(\overline{W}_0)]$) with the inverse of the Hessian. This multiplication with the inverse Hessian plays a key role in changing the direction of the steepest-gradient descent, so that one can take larger steps in that direction (resulting in better improvement of the objective function) even if the *instantaneous* rate of change in that direction is not as large as the steepest-descent direction. This is because the Hessian encodes how fast the gradient is changing in each direction. Changing gradients are bad for larger updates because one might inadvertently worsen the objective function, if the signs of many components of the gradient change during the step. It is profitable to move in directions where the ratio of the gradient to the rate of change of the gradient is large, so that one can take larger steps without causing harm to the optimization. Pre-multiplication with the inverse of the Hessian achieves this goal. The effect of the pre-multiplication of the steepest-descent direction with the inverse Hessian is shown in Figure 4.12. It is helpful to reconcile this figure with the example of the quadratic bowl in Figure 4.3. In a sense, pre-multiplication with the inverse Hessian biases the learning steps towards low-curvature directions. In one dimension, the Newton step is simply the ratio of the first derivative (rate of change) to the second derivative (curvature). In multiple dimensions, the low-curvature directions tend to win out because of multiplication by the inverse Hessian.

One interesting characteristic of the update of Equation 4.12 is that it is directly obtained from an optimality condition, and therefore there is no learning rate. In other words, this update is approximating the loss function with a quadratic bowl and moving *exactly* to the bottom of the bowl *in a single step*; the learning rate is already incorporated implicitly. However, when dealing with non-quadratic functions, it is useful to incorporate a learning rate and set it to minimize the objective function (and not just the quadratic

approximation). Therefore, for non-quadratic functions, the following iterative update is used:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - \alpha_t H^{-1}[\nabla L(\overline{W}_t)] \tag{4.13}$$

Here, α_t is chosen to minimize $L(\overline{W}_{t+1})$. This is done using a procedure called *line search*.

Assume that $\overline{q}_t = -H^{-1}\nabla L(\overline{W}_t)$ denotes the direction of movement in the Newton method. Then, Equation 4.13 an be written as follows:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$$

In line search, the step-size α_t is computed as follows:

$$\alpha_t = \text{argmin}_\alpha L(\overline{W}_t + \alpha \overline{q}_t) \tag{4.14}$$

There are several methods for performing line search, such as *binary search* and *golden section search*. In binary search, one starts by assuming that α_t belongs to the range $[a, b] = [0, \alpha_{max}]$, where α_{max} can be obtained by doubling successive intervals. Subsequently, one can reduce the size of the interval by a factor of 2 in each iteration by testing two points $c < d$ near the midpoint of the interval, and narrowing to the interval $[a, d]$ if the evaluation at $\alpha_t = d$ is larger. Otherwise the interval $[c, b]$ is used.

4.6.1 Example: Newton Method in the Quadratic Bowl

We will revisit how the Newton method behaves in the quadratic bowl of Figure 4.3. Consider the following elliptical objective function, which is the same as the one discussed in Figure 4.3(b):

$$J(w_1, w_2) = w_1^2 + 4w_2^2$$

This is a very simple convex quadratic, whose optimal point is the origin. Applying straightforward gradient descent starting at any point like $[w_1, w_2] = [1, 1]$ will result in the type of bouncing behavior shown in Figure 4.3(b). On the other hand, consider the Newton method, starting at the point $[w_1, w_2] = [1, 1]$. The gradient may be computed as $\nabla J(1, 1) = [2w_1, 8w_2]^T = [2, 8]^T$. Furthermore, the Hessian of this function is a constant that is independent of $[w_1, w_2]^T$:

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}$$

Applying the Newton update results in the following:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 2 \\ 8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

In other words, a single step suffices to reach the optimum point of this quadratic function. This is because the second-order Taylor "approximation" of a quadratic function is exact, and the Newton method solves this approximation in each iteration. Of course, real-world functions are not quadratic, and therefore multiple steps are typically needed.

4.6.2 Example: Newton Method in a Non-Quadratic Function

In this section, we will modify the objective function of the previous section to make it non-quadratic. The corresponding function is as follows:

$$J(w_1, w_2) = w_1^2 + 4w_2^2 - \cos(w_1 + w_2)$$

It is assumed that w_1 and w_2 are expressed[2] in radians. Note that the optimum of this objective function is still $[w_1, w_2] = [0, 0]$, since the value of $J(0, 0)$ is -1 at this point, where each additive term of the above expression takes on its minimum value. We will again start at $[w_1, w_2] = [1, 1]$, and show that one iteration no longer suffices in this case. In this case, we can show that the gradients and Hessian are as follows:

$$\nabla J(1,1) = \begin{bmatrix} 2 + \sin(2) \\ 8 + \sin(2) \end{bmatrix} = \begin{bmatrix} 2.91 \\ 8.91 \end{bmatrix}$$

$$H = \begin{bmatrix} 2 + \cos(2) & \cos(2) \\ \cos(2) & 8 + \cos(2) \end{bmatrix} = \begin{bmatrix} 1.584 & -0.416 \\ -0.416 & 7.584 \end{bmatrix}$$

The inverse of the Hessian is as follows:

$$H^{-1} = \begin{bmatrix} 0.64 & 0.035 \\ 0.035 & 0.134 \end{bmatrix}$$

Therefore, we obtain the following Newton update:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.64 & 0.035 \\ 0.035 & 0.134 \end{bmatrix} \begin{bmatrix} 2.91 \\ 8.91 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 2.1745 \\ 1.296 \end{bmatrix} = \begin{bmatrix} -1.1745 \\ -0.2958 \end{bmatrix}$$

Note that we do reach closer to an optimal solution, although we certainly do not reach the optimum point. This is because the objective function is not quadratic in this case, and one is only reaching the bottom of the *approximate* quadratic bowl of the objective function. However, Newton's method does find a better point in terms of the true objective function value. The approximate nature of the Hessian is why one must use either exact or approximate line search to control the step size. Note that if we used a step-size of 0.6 instead of the default value of 1, one would obtain the following solution:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.6 \begin{bmatrix} 2.1745 \\ 1.296 \end{bmatrix} = \begin{bmatrix} -0.30 \\ 0.22 \end{bmatrix}$$

Although this is only a very rough approximation to the optimal step size, it still reaches much closer to the true optimal value of $[w_1, w_2] = [0, 0]$. It is also relatively easy to show that this set of parameters yields a much better objective function value. This step would need to be repeated in order to reach closer and closer to an optimal solution.

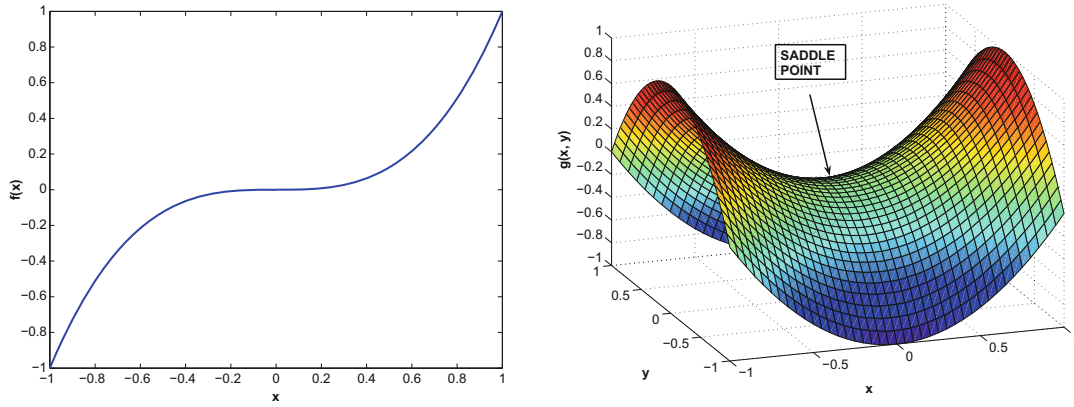
4.6.3 The Saddle-Point Problem with Second-Order Methods

Second-order methods are susceptible to the presence of *saddle points*. A saddle point is a stationary point of a gradient-descent method because its gradient is zero, but it is not a minimum (or maximum). A saddle point is an *inflection point*, which appears to be either a minimum or a maximum depending on which direction we approach it from. Therefore, the quadratic approximation of the Newton method will give vastly different shapes depending on the direction that one approaches the saddle point from. A 1-dimensional function with a saddle point is the following:

$$f(x) = x^3$$

This function is shown in Figure 4.13(a), and it has an inflection point at $x = 0$. Note that a quadratic approximation at $x > 0$ will look like an upright bowl, whereas a quadratic

[2]This ensures simplicity, as all calculus operations assume that angles are expressed in radians.



(a) 1-dimensional saddle point (b) 2-dimensional saddle point

Figure 4.13: Illustration of saddle points

approximation at $x < 0$ will look like an inverted bowl. Furthermore, even if one reaches $x = 0$ in the optimization process, both the second derivative and the first derivative will be zero. Therefore, a Newton update will take the 0/0 form and become indefinite. Such a point is a degenerate point from the perspective of numerical optimization. Not all saddle points are degenerate points and vice versa. For multivariate problems, such degenerate points are often wide and flat regions that are not minima of the objective function. They do present a significant problem for numerical optimization. An example of such a function is $h(x, y) = x^3 + y^3$, which is degenerate at $(0, 0)$. Furthermore, the region near $(0, 0)$ will appear like a flat plateau. These types of plateaus create problems for learning algorithms, because first-order algorithms slow down in these regions and second-order algorithms also cannot recognize them as unfruitful regions for exploration. It is noteworthy that such saddle points arise only in highly nonlinear functions, which are common in neural network optimization.

It is also instructive to examine the case of a saddle point that is not a degenerate point. An example of a 2-dimensional function with a saddle point is as follows:

$$g(x, y) = x^2 - y^2$$

This function is shown in Figure 4.13(b). The saddle point is $(0, 0)$. It is easy to see that the shape of this function resembles a riding saddle. In this case, approaching from the x direction or from the y direction will result in very different quadratic approximations. In one case, the function will appear to be a minimum, and in another case the function will appear to be a maximum. Furthermore, the saddle point $(0, 0)$ will be a stationary point from the perspective of a Newton update, even though it is not an extremum. Saddle points occur frequently in regions between two hills of the loss function, and they present a problematic topography for second-order methods. Interestingly, first-order methods are often able to escape from saddle points [153], because the trajectory of first-order methods is simply not attracted by such points. On the other hand, Newton's method will jump directly to the saddle point.

Unfortunately, some neural-network loss functions seem to contain a large number of saddle points. Second-order methods therefore are not always preferable to first-order methods; the specific topography of a particular loss function may have an important role to play.

Second-order methods are advantageous in situations with complex curvatures of the loss function or in the presence of cliffs. In other functions with saddle points, first-order methods are advantageous. Note that the pairing of computational algorithms (like Adam) with first-order methods can at least partially address many of the complexities in the loss surface such as plateaus or varying curvature. Therefore, real-world practitioners often prefer first-order methods in combination with computational algorithms like Adam. Recently, some methods have been proposed [91] to address saddle points in second-order methods.

4.7 Fast Approximations of Newton Method

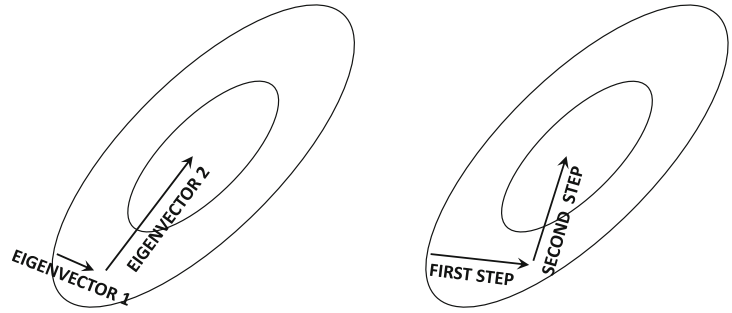
In most large-scale settings, the Hessian is too large to store or compute explicitly. It is not uncommon to have neural networks with millions of parameters. Trying to compute the inverse of a $10^6 \times 10^6$ Hessian matrix is a practically impossible task with the computational power available today. In fact, it is difficult to even compute the Hessian, let alone invert it! The following sections discuss efficient variations of the Newton method, which do not explicitly compute the Hessian.

4.7.1 Conjugate Gradient Method

The *conjugate gradient method* [199] requires d steps to reach the optimal solution of a quadratic loss function (instead of a single Newton step). The basic idea is that any quadratic function can be transformed to a linearly additive function by using an appropriate basis transformation of variables. These variables represent directions in the data that do not interact with one another. Such noninteracting directions are extremely convenient for optimization because they can be independently optimized with line search. Since it is possible to find such directions only for quadratic loss functions, we will first discuss the conjugate gradient method under the assumption that the loss function $L(\overline{W})$ is quadratic. Later, we will discuss the generalization to non-quadratic functions. This approach is well known in the classical literature on neural networks [41, 463], and a variant has recently been reborn under the title of "Hessian-free optimization." This name is motivated by the fact that the search direction can be computed without the explicit computation of the Hessian.

A quadratic and convex loss function $L(\overline{W})$ has an ellipsoidal contour plot of the type shown in Figure 4.14, and has a constant Hessian over all regions of the optimization space. The orthonormal eigenvectors $\overline{q}_0 \ldots \overline{q}_{d-1}$ of the symmetric Hessian represent the axes directions of the ellipsoidal contour plot. One can rewrite the loss function in a new coordinate space defined by the eigenvectors as the basis vectors *to create a linearly additive quadratic function in the different variables.* This is because the new coordinate system creates an basis-aligned ellipse, which does not have interacting quadratic terms of the type $x_i x_j$. Therefore, each transformed variable can be optimized independently of the others. Alternatively, one can work with the original variables (without transformation), and simply perform line search along each eigenvector of the Hessian to select the step size. The nature of the movement is illustrated in Figure 4.14(a). Note that movement along the jth eigenvector does not disturb the work done along other eigenvectors, and therefore d steps are sufficient to reach the optimal solution in quadratic loss functions.

Although it is impractical to compute the eigenvectors of the Hessian, there are other efficiently computable directions satisfying similar properties; this key property is referred to as *mutual conjugacy* of vectors. Note that the two eigenvectors $\overline{q}_i$ and $\overline{q}_j$ of the Hessian satisfy $\overline{q}_i^T \overline{q}_j = 0$ because of orthogonality of the eigenvectors of a symmetric matrix.



(a) Eigenvectors of Hessian Mutually Orthogonal: $\overline{q}_i^T \overline{q}_j = 0$

(b) Arbitrary conjugate pair Non-orthogonal: $\overline{q}_i^T H \overline{q}_j = 0$

Figure 4.14: The eigenvectors of the Hessian of a quadratic function represent the orthogonal axes of the quadratic ellipsoid and are also mutually orthogonal. The eigenvectors of the Hessian are orthogonal conjugate directions. The generalized definition of conjugacy may result in non-orthogonal directions.

Furthermore, since $\overline{q}_j$ is an eigenvector of H, we have $H\overline{q}_j = \lambda_j \overline{q}_j$ for some scalar eigenvalue λ_j. Multiplying both sides with $\overline{q}_i^T$, we can easily show that the eigenvectors of the Hessian satisfy $\overline{q}_i^T H \overline{q}_j = 0$ in pairwise fashion. The condition $\overline{q}_i^T H \overline{q}_j = 0$ is referred to as H-orthogonality in linear algebra, and is also referred to as the *mutual conjugacy condition* in optimization. It is this mutual conjugacy condition that results in linearly separated variables. However, the eigenvectors are not the only set of mutually conjugate conditions. Just as there are an infinite number of orthonormal basis sets, there are an infinite number of H-orthogonal basis sets in d-dimensional space. If we re-write the quadratic loss function in terms of coordinates in a *any* (possibly non-orthogonal) axis system of conjugate directions, the objective function will contain linearly additive components each of which is a univariate function. One can separately optimize along each of these d H-orthogonal directions (in terms of the original variables) to solve the quadratic optimization problem in d steps. Each of these optimization steps can be performed using line search along an H-orthogonal direction. Hessian eigenvectors represent a rather special set of H-orthogonal directions that are also orthogonal; conjugate directions other than Hessian eigenvectors, such as those shown in Figure 4.14(b), are not mutually orthogonal. Therefore, conjugate gradient descent optimizes a quadratic objective function by *implicitly* transforming the loss function into a *non-orthogonal* basis with a linearly additive representation of the objective function. One can state this observation as follows:

Observation 4.7.1 (Properties of H-Orthogonal Directions) *Let H be the Hessian of a quadratic objective function. If any set of d H-orthogonal directions are selected for movement, then one is implicitly moving along additively separable variables in a transformed representation of the function. Therefore, at most d steps are required for quadratic optimization.*

The independent optimization along each separable direction (with line search) ensures that the component of the gradient along each conjugate direction will be 0. Strictly convex loss functions have linearly independent conjugate directions (see Exercise 5). In other words, the final gradient will have zero dot product with d linearly independent directions; this is possible only when the final gradient is the zero vector (see Exercise 6), which implies

optimality for a convex function. In fact, one can often reach a near-optimal solution in far fewer than d updates.

How can one identify conjugate directions? The simplest approach is to use *generalized Gram-Schmidt orthogonalization* on the Hessian of the quadratic function in order to generate H-orthogonal directions (see Exercise 7). Such an orthogonalization is easy to achieve using arbitrary vectors as starting points. However, this process can still be quite expensive because each direction $\overline{q}_t$ needs to use *all* the previous directions $\overline{q}_0 \ldots \overline{q}_{t-1}$ for iterative generation in the Gram-Schmidt method. Since each direction is a d-dimensional vector, and there are $O(d)$ such directions towards the end of the process, it follows that each step will require $O(d^2)$ time. Is there a way to do this using only the previous direction in order to reduce this time from $O(d^2)$ to $O(d)$? Surprisingly, only the most recent conjugate direction is needed to generate the next direction [370, 463], when steepest descent directions are used for iterative generation. In other words, one should not use Gram-Schmidt orthogonalization with arbitrary vectors, but should use steepest descent directions as the raw vectors to be orthogonalized. This choice makes all the difference in ensuring a more efficient form of orthogonalization. This is not an obvious result (see Exercise 8). The direction $\overline{q}_{t+1}$ is, therefore, defined iteratively as a linear combination of *only* the previous conjugate direction $\overline{q}_t$ and the current steepest descent direction $\nabla L(\overline{W}_{t+1})$ with combination parameter β_t:

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \beta_t \overline{q}_t \tag{4.15}$$

Premultiplying both sides with $\overline{q}_t^T H$ and using the conjugacy condition to set the left-hand side to 0, one can solve for β_t:

$$\beta_t = \frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \tag{4.16}$$

This leads to an iterative update process, which initializes $\overline{q}_0 = -\nabla L(\overline{W}_0)$, and computes $\overline{q}_{t+1}$ iteratively for $t = 0, 1, 2, \ldots T$:

1. Update $\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$. Here, the step size α_t is computed using line search to minimize the loss function.
2. Set $\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t$. Increment t by 1.

It can be shown [370, 463] that $\overline{q}_{t+1}$ satisfies conjugacy with respect to *all* previous $\overline{q}_i$. A systematic road-map of this proof is provided in Exercise 8.

The above updates do not *seem* to be Hessian-free, because the matrix H is included in the above updates. However, the underlying computations only need the *projection* of the Hessian along particular directions; we will see that these can be computed indirectly using the method of finite differences without explicitly computing the individual elements of the Hessian. Let $\overline{v}$ be the vector direction for which the projection $H\overline{v}$ needs to be computed. The method of finite differences computes the loss gradient at the current parameter vector $\overline{W}$ and at $\overline{W} + \delta\overline{v}$ for some small value of δ in order to perform the approximation:

$$H\overline{v} \approx \frac{\nabla L(\overline{W} + \delta\overline{v}) - \nabla L(\overline{W})}{\delta} \propto \nabla L(\overline{W} + \delta\overline{v}) - \nabla L(\overline{W}) \tag{4.17}$$

The right-hand side is free of the Hessian. The condition is exact for quadratic functions. Other alternatives for Hessian-free updates are discussed in [41].

So far, we have discussed the simplified case of quadratic loss functions, in which the Hessian is a constant matrix (i.e., independent of the current parameter vector). However,

most loss functions in machine learning are not quadratic and, therefore, the Hessian matrix is dependent on the current value of the parameter vector $\overline{W}_t$. This leads to several choices in terms of how one can create a modified algorithm for non-quadratic functions. Do we first create a quadratic approximation at a point and then solve it for a few iterations with the Hessian (quadratic approximation) fixed at that point, or do we change the Hessian every iteration along with the change in parameter vector? The former is referred to as the *linear conjugate gradient method*, whereas the latter is referred to as the *nonlinear conjugate gradient method*.

In the nonlinear conjugate gradient method, the mutual conjugacy (i.e., H-orthogonality) of the directions will deteriorate over time, as the Hessian changes from one step to the next. This can have an unpredictable effect on the overall progress from one step to the next. Furthermore, the computation of conjugate directions needs to be restarted every few steps, as the mutual conjugacy deteriorates. If the deterioration occurs too fast, the restarts occur very frequently, and one does not gain much from conjugacy. On the other hand, each quadratic approximation in the linear conjugate gradient method can be solved exactly, and will typically be (almost) solved in much fewer than d iterations. Therefore, one can make similar progress to the Newton method in each iteration. As long as the quadratic approximation is of high quality, the required number of approximations is often not too large. The nonlinear conjugate gradient method has been extensively used in traditional machine learning from a historical perspective [41], although recent work [324, 325] has advocated the use of linear conjugate methods. Experimental results in [324, 325] suggest that linear conjugate gradient methods have some advantages.

4.7.2 Quasi-Newton Methods and BFGS

The acronym BFGS stands for the Broyden–Fletcher–Goldfarb–Shanno algorithm, and it is derived as an approximation of the Newton method. Let us revisit the updates of the Newton method. A typical update of the Newton method is as follows:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla J(\overline{W}_0)] \tag{4.18}$$

In quasi-Newton methods, a sequence of approximations of the inverse Hessian matrix are used in various steps. Let the approximation of the inverse Hessian matrix in the tth step be denoted by G_t. In the very first iteration, the value of G_t is initialized to the identity matrix, which amounts to moving along the steepest-descent direction. This matrix is continuously updated from G_t to G_{t+1} with low-rank updates (derived from the matrix inversion lemma of Chapter 1). A direct restatement of the Newton update in terms of the inverse Hessian $G_t \approx H_t^{-1}$ is as follows:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - G_t[\nabla J(\overline{W}_t)] \tag{4.19}$$

The above update can be improved with an optimized learning rate α_t for non-quadratic loss functions working with (inverse) Hessian approximations like G_t:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - \alpha_t G_t[\nabla J(\overline{W}_t)] \tag{4.20}$$

The optimized learning rate α_t is identified with line search. The line search does not need to be performed exactly (like the conjugate gradient method), because maintenance of conjugacy is no longer critical. Nevertheless, approximate conjugacy of the early set of directions is maintained by the method when starting with the identity matrix. One can (optionally) reset G_t to the identity matrix every d iterations (although this is rarely done).

It remains to be discussed how the matrix G_{t+1} is approximated from G_t. For this purpose, the *quasi-Newton condition*, also referred to as the *secant condition*, is needed:

$$\underbrace{\overline{W}_{t+1} - \overline{W}_t}_{\text{Parameter Change}} = G_{t+1} \underbrace{[\nabla J(\overline{W}_{t+1}) - \nabla J(\overline{W}_t)]}_{\text{First derivative change}} \tag{4.21}$$

The above formula is simply a finite-difference approximation. Intuitively, multiplication of the second-derivative matrix (i.e., Hessian) with the parameter change (vector) approximately provides the gradient change. Therefore, multiplication of the inverse Hessian approximation G_{t+1} with the gradient change provides the parameter change. The goal is to find a symmetric matrix G_{t+1} satisfying Equation 4.21, but it represents an underdetermined system of equations with an infinite number of solutions. Among these, BFGS chooses the closest symmetric G_{t+1} to the current G_t, and achieves this goal by posing a minimization objective function $||G_{t+1} - G_t||_w$ in the form of a *weighted Frobenius norm*. In other words, we want to find G_{t+1} satisfying the following:

$$\begin{aligned} \text{Minimize}_{[G_{t+1}]} \; & ||G_{t+1} - G_t||_w \\ & \text{subject to:} \\ & \overline{W}_{t+1} - \overline{W}_t = G_{t+1}[\nabla J(\overline{W}_{t+1}) - \nabla J(\overline{W}_t)] \\ & G_{t+1}^T = G_{t+1} \end{aligned}$$

The subscript of the norm is annotated by "w" to indicate that it is a weighted[3] form of the norm. This weight is an "averaged" form of the Hessian, and we refer the reader to [370] for details of how the averaging is done. Note that one is not constrained to using the weighted Frobenius norm, and different variations of how the norm is constructed lead to different variations of the quasi-Newton method. For example, one can pose the same objective function and secant condition in terms of the Hessian rather than the inverse Hessian, and the resulting method is referred to as the Davidson–Fletcher–Powell (DFP) method. In the following, we will stick to the use of the inverse Hessian, which is the BFGS method.

Since the weighted norm uses the Frobenius matrix norm (along with a weight matrix) the above is a quadratic optimization problem with linear constraints. In general, when there are linear equality constraints paired with a quadratic objective function, the structure of the optimization problem is quite simple, and closed-form solutions can sometimes be found. This is because the equality constraints can often be eliminated along with corresponding variables (using methods like Gaussian elimination), and an unconstrained, quadratic optimization problem can be defined in terms of the remaining variables. These problems sometimes turn out to have closed-form solutions like least-squared regression. In this case, the closed-form solution to the above optimization problem is as follows:

$$G_{t+1} \Leftarrow (I - \Delta_t \overline{q}_t \overline{v}_t^T) G_t (I - \Delta_t \overline{v}_t \overline{q}_t^T) + \Delta_t \overline{q}_t \overline{q}_t^T \tag{4.22}$$

Here, the (column) vectors $\overline{q}_t$ and $\overline{v}_t$ represent the parameter change and the gradient change; the scalar $\Delta_t = 1/(\overline{q}_t^T \overline{v}_t)$ is the inverse of the dot product of these two vectors.

$$\overline{q}_t = \overline{W}_{t+1} - \overline{W}_t; \quad \overline{v}_t = \nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)$$

[3]The form of the objective function is $||A^{1/2}(G_{t+1} - G_t)A^{1/2}||_F$ norm, where A is an averaged version of the Hessian matrix over various lengths of the step. We refer the reader to [370] for details.

The update in Equation 4.22 can be made more space efficient by expanding it, so that fewer temporary matrices need to be maintained. Interested readers are referred to [309, 370, 390] for implementation details and derivation of these updates.

Even though BFGS benefits from approximating the inverse Hessian, it does need to carry over a matrix G_t of size $O(d^2)$ from one iteration to the next. The *limited memory BFGS* (L-BFGS) reduces the memory requirement drastically from $O(d^2)$ to $O(d)$ by not carrying over the matrix G_t from the previous iteration. In the most basic version of the L-BFGS method, the matrix G_t is replaced with the identity matrix in Equation 4.22 in order to derive G_{t+1}. A more refined choice is to store the $m \approx 30$ most recent vectors $\overline{q}_t$ and $\overline{v}_t$. Then, L-BFGS is equivalent to initializing G_{t-m+1} to the identity matrix and recursively applying Equation 4.22 m times to derive G_{t+1}. In practice, the implementation is optimized to directly compute the direction of movement from the vectors without explicitly storing large intermediate matrices from G_{t-m+1} to G_t.

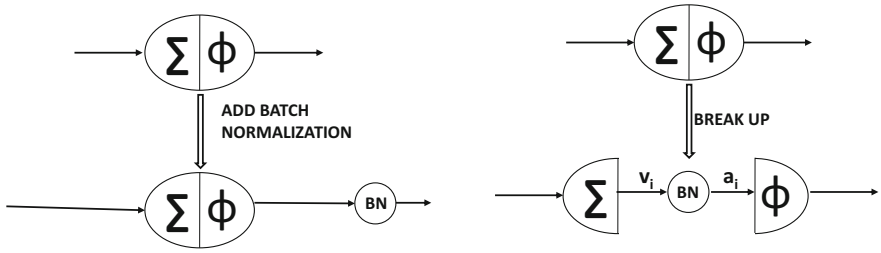
4.8 Batch Normalization

Batch normalization is a recent method to address the vanishing and exploding gradient problems, which cause activation gradients in successive layers to either reduce or increase in magnitude. Another important problem in training deep networks is that of *internal covariate shift*. The problem is that the parameters change during training, and therefore the hidden variable activations change as well. In other words, the hidden inputs from early layers to later layers keep changing. Changing inputs from early layers to later layers causes slower convergence during training because the training data for later layers is not stable. Batch normalization is able to reduce this effect.

In batch normalization, the idea is to add additional "normalization layers" between hidden layers that resist this type of behavior by creating features with somewhat similar variance. Furthermore, each unit in the normalization layers contains two additional parameters β_i and γ_i that regulate the precise level of normalization in the ith unit; these parameters are learned in a data-driven manner. The basic idea is that the output of the ith unit will have a mean of β_i and a standard deviation of γ_i *over each mini-batch of training instances*. One might wonder whether it might make sense to simply set each β_i to 0 and each γ_i to 1, but doing so reduces the representation power of the network. For example, if we make this transformation, then the sigmoid units will be operating within their linear regions, especially if the normalization is performed just before activation (see below for discussion of Figure 4.15). Recall from the discussion in Chapter 1 that multilayer networks do not gain power from depth without nonlinear activations. Therefore, allowing some "wiggle" with these parameters and learning them in a data-driven manner makes sense. Furthermore, the parameter β_i plays the role of a learned bias variable, and therefore we do not need additional bias units in these layers.

We assume that the ith unit is connected to a special type of node BN_i, where BN stands for batch normalization. This unit contains two parameters β_i and γ_i that need to be learned. Note that BN_i has only one input, and its job is to perform the normalization and scaling. This node is then connected to the next layer of the network in the standard way in which a neural network is connected to future layers. Here, we mention that there are two choices for where the normalization layer can be connected:

1. The normalization can be performed just after applying the activation function to the linearly transformed inputs. This solution is shown in Figure 4.15(a). Therefore, the normalization is performed on *post-activation values*.



(a) Post-activation normalization (b) Pre-activation normalization

Figure 4.15: The different choices in batch normalization

2. The normalization can be performed after the linear transformation of the inputs, but before applying the activation function. This situation is shown in Figure 4.15(b). Therefore, the normalization is performed on *pre-activation values*.

It is argued in [225] that the second choice has more advantages. Therefore, we focus on this choice in this exposition. The BN node shown in Figure 4.15(b) is just like any other computational node (albeit with some special properties), and one can perform backpropagation through this node just like any other computational node.

What transformations does BN_i apply? Consider the case in which its input is $v_i^{(r)}$, corresponding to the rth element of the batch feeding into the ith unit. Each $v_i^{(r)}$ is obtained by using the linear transformation defined by the coefficient vector $\overline{W}_i$ (and biases if any). For a particular batch of m instances, let the values of the m activations be denoted by $v_i^{(1)}, v_i^{(2)}, \ldots v_i^{(m)}$. The first step is to compute the mean μ_i and standard deviation σ_i for the ith hidden unit. These are then scaled using the parameters β_i and γ_i to create the outputs for the next layer:

$$\mu_i = \frac{\sum_{r=1}^{m} v_i^{(r)}}{m} \quad \forall i \tag{4.23}$$

$$\sigma_i^2 = \frac{\sum_{r=1}^{m} (v_i^{(r)} - \mu_i)^2}{m} + \epsilon \quad \forall i \tag{4.24}$$

$$\hat{v}_i^{(r)} = \frac{v_i^{(r)} - \mu_i}{\sigma_i} \quad \forall i, r \tag{4.25}$$

$$a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i \quad \forall i, r \tag{4.26}$$

A small value of ϵ is added to σ_i^2 to regularize cases in which all activations are the same, which results in zero variance. Note that $a_i^{(r)}$ is the pre-activation output of the ith node, when the rth batch instance passes through it. This value would otherwise have been set to $v_i^{(r)}$, if we had not applied batch normalization. We conceptually represent this node with a special node BN_i that performs this additional processing. This node is shown in Figure 4.15(b). Therefore, the backpropagation algorithm has to account for this additional node and ensure that the loss derivative of layers earlier than the batch normalization layer accounts for the transformation implied by these new nodes. It is important to note that the function applied at each of these special BN nodes is specific to the *batch* at hand. This type of computation is unusual for a neural network in which the gradients are linearly additive sums of the gradients with respect to individual training examples. This is not quite true in

this case because the batch normalization layer computes nonlinear metrics from the batch (such as its standard deviation). Therefore, the activations depend on how the examples in a batch are related to one another, which is not common in most neural computations. However, this special property of the BN node does not prevent us from backpropagating through the computations performed in it.

The following will describe the changes in the backpropagation algorithm caused by the normalization layer. The main point of this change is to show how to backpropagate through the newly added layer of normalization nodes. Another point to be aware of is that we want to optimize the parameters β_i and γ_i. For the gradient-descent steps with respect to each β_i and γ_i, we need the gradients with respect to these parameters. Assume that we have already backpropagated up to the output of the BN node, and therefore we have each $\frac{\partial L}{\partial a_i^{(r)}}$ available. Then, the derivatives with respect to the two parameters can be computed as follows:

$$\frac{\partial L}{\partial \beta_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \beta_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}}$$

$$\frac{\partial L}{\partial \gamma_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \gamma_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \hat{v}_i^{(r)}$$

We also need a way to compute $\frac{\partial L}{\partial v_i^r}$. Once this value is computed, the backpropagation to the pre-activation values $\frac{\partial L}{\partial a_j^r}$ for all nodes j in the previous layer uses the straightforward backpropagation update introduced earlier in this chapter. Therefore, the dynamic programming recursion will be complete because one can then use these values of $\frac{\partial L}{\partial a_j^r}$. One can compute the value of $\frac{\partial L}{\partial v_i^r}$ in terms of $\hat{v}_i^{(r)}$, μ_i, and σ_i, by observing that $v_i^{(r)}$ can be written as a (normalization) function of only $\hat{v}_i^{(r)}$, mean μ_i, and variance σ_i^2. Observe that μ_i and σ_i are not treated as constants, but as variables because they depend on the batch at hand. Therefore, we have the following:

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial \hat{v}_i^{(r)}} \frac{\partial \hat{v}_i^{(r)}}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial v_i^{(r)}} \tag{4.27}$$

$$= \frac{\partial L}{\partial \hat{v}_i^{(r)}} \left(\frac{1}{\sigma_i}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(v_i^{(r)} - \mu_i)}{m}\right) \tag{4.28}$$

We need to evaluate each of the three partial derivatives on the right-hand side of the above equation in terms of the quantities that have been computed using the already-executed dynamic programming updates of backpropagation. This allows the creation of the recurrence equation for the batch normalization layer. Among these, the first expression, which is $\frac{\partial L}{\partial \hat{v}_i^{(r)}}$, can be substituted in terms of the loss derivatives of the next layer by observing that $a_i^{(r)}$ is related to $\hat{v}_i^{(r)}$ by a constant of proportionality γ_i:

$$\frac{\partial L}{\partial \hat{v}_i^{(r)}} = \gamma_i \frac{\partial L}{\partial a_i^{(r)}} \quad [\text{Since } a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i] \tag{4.29}$$

Therefore, by substituting this value of $\frac{\partial L}{\partial \hat{v}_i^{(r)}}$ in Equation 4.28, we have the following:

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial a_i^{(r)}} \left(\frac{\gamma_i}{\sigma_i}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(v_i^{(r)} - \mu_i)}{m}\right) \tag{4.30}$$

It now remains to compute the partial derivative of the loss with respect to the mean and the variance. The partial derivative of the loss with respect to the variance is computed as follows:

$$\frac{\partial L}{\partial \sigma_i^2} = \underbrace{\sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} \cdot \frac{\partial \hat{v}_i^{(q)}}{\partial \sigma_i^2}}_{\text{Chain rule}} = \underbrace{-\frac{1}{2\sigma_i^3} \sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} (v_i^{(q)} - \mu_i)}_{\text{Use Equation 4.25}} = \underbrace{-\frac{1}{2\sigma_i^3} \sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}} \gamma_i \cdot (v_i^{(q)} - \mu_i)}_{\text{Substitution from Equation 4.29}}$$

The partial derivatives of the loss with respect to the mean can be computed as follows:

$$\frac{\partial L}{\partial \mu_i} = \underbrace{\sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} \cdot \frac{\partial \hat{v}_i^{(q)}}{\partial \mu_i} + \frac{\partial L}{\partial \sigma_i^2} \cdot \frac{\partial \sigma_i^2}{\partial \mu_i}}_{\text{Chain rule}} = \underbrace{-\frac{1}{\sigma_i} \sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} - 2\frac{\partial L}{\partial \sigma_i^2} \cdot \frac{\sum_{q=1}^{m} (v_i^{(q)} - \mu_i)}{m}}_{\text{Use Equations 4.24 and 4.25}}$$

$$= \underbrace{-\frac{\gamma_i}{\sigma_i} \sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}}}_{\text{Eq. 4.29}} + \underbrace{\left(\frac{1}{\sigma_i^3}\right) \cdot \left(\sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}} \gamma_i \cdot (v_i^{(q)} - \mu_i)\right) \cdot \left(\frac{\sum_{q=1}^{m} (v_i^{(q)} - \mu_i)}{m}\right)}_{\text{Substitution for } \frac{\partial L}{\partial \sigma_i^2}}$$

By plugging in the partial derivatives of the loss with respect to the mean and variance in Equation 4.30, we get a full recursion for $\frac{\partial L}{\partial v_i^{(r)}}$ (value before batch-normalization layer) in terms of $\frac{\partial L}{\partial a_i^{(r)}}$ (value *after* the batch normalization layer). This provides a full view of the backpropagation of the loss through the batch-normalization layer corresponding to the BN node. The other aspects of backpropagation remain similar to the traditional case. Batch normalization enables faster inference because it prevents problems such as the exploding and vanishing gradient (which cause slow learning).

A natural question about batch normalization arises during inference (prediction) time. Since the transformation parameters μ_i and σ_i depend on the batch, how should one compute them during testing when a *single* test instance is available? In this case, the values of μ_i are σ_i are computed up front using the *entire* population (of training data), and then treated as constants during testing time. One can also keep an exponentially weighted average of these values during training. Therefore, the normalization is a simple linear transformation during inference.

An interesting property of batch normalization is that *it also acts as a regularizer*. Note that the same data point can cause somewhat different updates depending on which batch it is included in. One can view this effect as a kind of noise added to the update process. Regularization is often achieved by adding a small amount of noise to the training data. It has been experimentally observed that regularization methods like *Dropout* (cf. section 5.5.4 of Chapter 5) do not seem to improve performance when batch normalization is used [194], although there is not a complete agreement on this point. A variant of batch normalization, known as *layer normalization*, is known to work well with recurrent networks. This approach is discussed in section 8.3.1 of Chapter 8.

4.9 Practical Tricks for Acceleration and Compression

Neural network learning algorithms can be extremely expensive, both in terms of the number of parameters in the model and the amount of data that needs to be processed. There are

several strategies that are used to accelerate and compress the underlying implementations. Some of the common strategies are as follows:

1. *GPU-acceleration:* Graphics Processor Units (GPUs) have historically been used for rendering video games with intensive graphics because of their efficiency in settings where repeated matrix operations (e.g., on graphics pixels) are required. It was eventually realized by the machine learning community (and GPU hardware companies) that such repetitive matrix operations are also used in deep learning. Even the use of a single GPU can significantly speed up implementation because of its high memory bandwidth and multithreading within its multicore architecture.

2. *Parallel implementations:* One can parallelize the implementations of neural networks by using multiple GPUs or CPUs. Either the neural network model or the data can be partitioned across different processors. These implementations are referred to as *model-parallel* and *data-parallel* implementations.

3. *Algorithmic tricks for model compression:* A key point about the practical use of neural networks is that they have different computational requirements during training and deployment (prediction). While it is acceptable to train a model for a week on state-of-the-art hardware, the final deployment might be performed on a mobile phone with limited memory and computational power. Therefore, numerous tricks are used for model compression during testing time. This type of compression often results in better cache performance and efficiency as well.

In the following, we will discuss some of these acceleration and compression techniques.

4.9.1 GPU Acceleration

GPUs were originally developed for rendering graphics on screens with the use of lists of 3-dimensional coordinates. Therefore, graphics cards were inherently designed to perform many matrix multiplications in parallel to render the graphics rapidly. GPU processors have evolved significantly, moving well beyond their original functionality of graphics rendering. Like graphics applications, neural-network implementations require large matrix multiplications, which is inherently suited to the GPU setting. In a traditional neural network, each forward propagation is a multiplication of a matrix and vector, whereas in a convolutional neural network, two matrices are multiplied. When a mini-batch approach is used, activations become matrices (instead of vectors) in a traditional neural network. Therefore, forward propagations require matrix multiplications. A similar result is true for backpropagation, during which two matrices are multiplied frequently to propagate the derivatives backwards. In other words, most of the intensive computations involve vector, matrix, and tensor operations. Even a single GPU is good at parallelizing these operations in its different cores with multithreading [213], in which some groups of threads sharing the same code are executed concurrently. This principle is referred to as *Single Instruction Multiple Threads (SIMT)*. Although CPUs also support short-vector data parallelization via *Single Instruction Multiple Data (SIMD)* instructions, the degree of parallelism is much lower as compared to the GPU. There are different trade-offs when using GPUs as compared to traditional CPUs. GPUs are very good at repetitive operations, but they have difficulty at performing branching operations like *if-then* statements. Most of the intensive operations in neural network learning are repetitive matrix multiplications across different training instances, and therefore this setting is suited to the GPU. Although the clock speed of a

single instruction in the GPU is slower than the traditional CPU, the parallelization is so much greater in the GPU that huge advantages are gained.

GPU threads are grouped into small units called *warps*. Each thread in the warp shares the same code in each cycle, and this restriction enables a concurrent execution of the threads. The implementation needs to be carefully tailored to reduce the use of memory bandwidth. This is done by *coalescing* the memory reads and writes from different threads, so that a single memory transaction can be used to read and write values from different threads. Consider a common operation like matrix multiplication in neural network settings. The matrices are multiplied by making each thread responsible for computing a single entry in the product matrix. For example, consider a situation in which a 100×50 matrix is multiplied with a 50×200 matrix. In such a case, a total of $100 \times 200 = 20000$ threads would be launched in order to compute the entries of the matrix. These threads will typically be partitioned into multiple warps, each of which is highly parallelized. Therefore, speedups are achieved. A discussion of matrix multiplication on GPUs is provided in [213].

With high amounts of parallelization, memory bandwidth is often the primary limiting factor. Memory bandwidth refers to the speed at which the processor can access the relevant parameters from their stored locations in memory. GPUs have a high degree of parallelism and high memory bandwidth as compared to traditional CPUs. Note that if one cannot access the relevant parameters from memory fast enough, then faster execution does not help the speed of computation. In such cases, the memory transfer cannot keep up with the speed of the processor whether working with the CPU or the GPU, and the CPU/GPU cores will idle. GPUs have different trade-offs between cache access, computation, and memory access. CPUs have much larger caches than GPUs and they rely on the caches to store an intermediate result, such as the result of multiplying two numbers. Accessing a computed value from a cache is much faster than multiplying them again, which is where the CPU has an advantage over the GPU. However, this advantage is neutralized in neural network settings, where the sizes of the parameter matrices and activations are often too large to fit in the CPU cache. Even though the CPU cache is larger than that of the GPU, it is not large enough to handle the scale at which neural-network operations are performed. In such cases, one has to rely on high memory bandwidth, which is where the GPU has an advantage over the CPU. Furthermore, it is often faster to perform the same computation again rather than accessing it from memory, when working with the GPU (assuming that the result is unavailable in a cache). Therefore, GPU implementations are done somewhat differently from traditional CPU implementations. Furthermore, the advantage gained can be sensitive to the choice of neural network architecture, as the memory bandwidth requirements and multi-threading gains of different architectures can be different.

At first sight, it might seem that a lot of challenging and customized low-level programming is required for each GPU-based neural architecture. With this problem in mind, companies like NVIDIA have modularized the programmer interface with the GPU implementation. The key point is that the speeding of primitives like matrix multiplication and convolution can be hidden from the user by providing a library of neural network operations that perform these faster operations behind the scenes. The GPU library is tightly integrated with deep learning frameworks like Caffe or Torch to take advantage of accelerated GPU operations. A specific example of such a library is the *NVIDIA CUDA Deep Neural Network Library* [664], which is referred to in short as *cuDNN*. CUDA is a parallel computing platform and programming model that works with CUDA-enabled GPU processors. However, it provides an abstraction and a programming interface that is easy to use with relatively limited rewriting of code. The cuDNN library can be integrated with multiple deep learning frameworks such as Caffe, TensorFlow, Theano, and Torch. The changes re-

quired to convert the training code of a particular neural network from its CPU version to a GPU version are often small. For example, in Torch, the CUDA Torch package is incorporated at the beginning of the code, and various data structures (like tensors) are initialized as CUDA tensors (instead of regular tensors). With these types of modest modifications, virtually the same code can run on a GPU instead of a CPU in Torch. A similar situation holds true in other deep learning frameworks. Such an approach shields developers from the low-level performance tuning required in GPU frameworks, because the implementations of the library primitives take care of all the low-level details of GPU parallelization.

4.9.2 Parallel and Distributed Implementations

It is possible to make training even faster by using multiple CPUs or GPUs. Since it is more common to use multiple GPUs, we focus on this setting. Parallelism is not a simple matter when working with GPUs because there are overheads associated with the communication between different processors. The delay caused by these overheads has recently been reduced with specialized network cards for GPU-to-GPU transfer. Furthermore, algorithmic tricks like using 8-bit approximations of the gradients [100] can help in speeding up the communication. There are several ways in which one can partition the work across different processors, namely hyperparameter parallelism, model parallelism, and data parallelism. These methods are discussed below.

Hyperparameter Parallelism

The simplest possible way to achieve parallelism in the training process without much overhead is to train neural networks with different parameter settings on different processors. No communication is required across different executions, and therefore wasteful overhead is avoided. As discussed earlier in this chapter, runs with suboptimal hyperparameters are often terminated long before running them to completion. Nevertheless, a small number of different runs with optimized parameters are often used in order to create an ensemble of models. The training of different ensemble components can be performed independently on different processors.

Model Parallelism

Model parallelism is particularly useful when a single model is too large to fit on a GPU. In such a case, the hidden layer is divided across the different GPUs. The different GPUs work on exactly the same batch of training points, although different GPUs compute different parts of the activations and the gradients. Each GPU only contains the portion of the weight matrix that are multiplied with the hidden activations present in the GPU. However, it would still need to communicate the results of its activations to the other GPUs. Similarly, it would need to receive the derivatives with respect to the hidden units in other GPUs in order to compute the gradients of the weights between its hidden units and those of other GPUs. This is achieved with the use of inter-connections across GPUs, and the computations across these interconnections add to the overhead. In some cases, these interconnections are dropped in a subset of the layers in order to reduce the communication overhead (although the resulting model would not quite be the same as the sequential version). Model parallelism is not helpful in cases where the number of parameters in the neural network is small, and should only be used for large networks. A good practical example of model parallelism is the design of *AlexNet*, which is a convolutional neural network (cf. section 9.4.1 of Chapter 9). A sequential version of *AlexNet* and a GPU-partitioned version of *AlexNet* are both shown

in Figure 9.9 of Chapter 9. Note that the sequential version in Figure 9.9 is not exactly equivalent to the GPU-partitioned version because the interconnections between GPUs have been dropped in some of the layers. A discussion of model parallelism may be found in [77].

Data Parallelism

Data parallelism works best when the model is small enough to fit on each GPU, but the amount of training data is large. In these cases, the parameters are shared across the different GPUs and the goal of the updates is to use the different processors with different training points for faster updates. The problem is that perfect synchronization of the updates can slow down the process, because locking mechanisms would need to be used to synchronize the updates. The key point is that each processor would have to wait for the others to make their updates. As a result, the slowest processor creates a bottleneck. A method that uses *asynchronous* stochastic gradient descent was proposed in [94]. The basic idea is to use a parameter server in order to share the parameters across different GPU processors. The updates are performed without using any locking mechanism. In other words, each GPU can read the shared parameters at any time, perform the computation, and write the parameters to the parameter server without worrying about locks. In this case, inefficiency would still be caused by one GPU processor overwriting the progress made by another, but there would be no waiting times for writes. As a result, the overall progress would still be faster than with a synchronized mechanism. Distributed asynchronous gradient descent is quite popular as a strategy for parallelism in large-scale industrial settings.

Exploiting the Trade-Offs for Hybrid Parallelism

It is evident from the above discussion that model parallelism is well suited to models with a large parameter footprint, whereas data parallelism is well suited to smaller models. It turns out that one can combine the two types of parallelism over different parts of the network. In certain types of convolutional neural networks that have fully connected layers, the vast majority of parameters occur in the fully connected layers, whereas more computations are performed in the earlier layers. In these cases, it makes sense to use data parallelism for the early part of the network, and model parallelism for the later part of the network. This type of approach is referred to as *hybrid parallelism*. A discussion of this type of approach may be found in [262].

4.9.3 Algorithmic Tricks for Model Compression

Training a neural network and deploying it typically have different requirements in terms of memory and efficiency requirements. While it may be acceptable to require a week to train a neural network to recognize faces in images, the end user might wish to use the trained neural network to recognize a face within a a few seconds. Furthermore, the model might be deployed on a mobile device with limited memory and computational availability. Computational efficiency is generally not a problem at deployment time, because the prediction of a test instance often requires straightforward matrix multiplications over a few layers. On the other hand, the storage requirements are often a problem because of the large number of parameters in multilayer networks. There are several tricks that are used for model compression in such cases. In most of the cases, a larger trained neural network is modified so that it requires less space by approximating some parts of the model. In addition, some efficiency improvements can also be realized at prediction time by model

compression because of better cache performance and fewer operations, although this is not the primary goal. Interestingly, this approximation might occasionally *improve* accuracy on out-of-sample predictions because of regularization effects, especially if the original model is unnecessarily large compared to the training data size.

Pruning Network Weights

The links in a neural network are associated with weights. If the absolute value of a particular weight is small, then the model is not strongly influenced by that weight. Such weights can be dropped, and the neural network can be fine-tuned starting with the current weights on links that have not yet been dropped. By repeating the process of network pruning and fine-tuning iteratively, it is possible to improve compression effects. One can also encourage the dropping of links by using L_1-regularization, because it causes many learned weights to have zero values (see Chapter 5). However, L_2-regularization (with pruning of low weights) seems to be preferred because of higher accuracy [179]. An interesting recent result, referred to as the *lottery ticket hypothesis* [126], shows that instead of fine-tuning the learned weights, using the *initialization* weights from the previous round of training to retrain the network provides excellent results (while random initialization on the pruned network does poorly). The basic idea is that neural networks need to have massively redundant structures for achieving high accuracy and appropriate subnetworks are implicitly chosen by specific initializations. For modest levels of pruning, accuracy might *improve* because of regularization effects.

Further enhancements were reported in [178], where the approach was combined with Huffman coding and quantization for compression. The goal of quantization is to reduce the number of bits representing each connection. This approach reduced the storage required by *AlexNet* [263] by a factor of 35, or from about 240MB to 6.9MB, with no loss of accuracy. It is now possible as a result of this reduction to fit the model into an on-chip SRAM cache rather than off-chip DRAM memory; this also provide a beneficial effect on prediction times.

Compressing Network Weights

It was shown in [96] that the vast majority of the weights in a neural network are redundant. In other words, for any $m \times n$ weight matrix W between a pair of layers with m_1 and m_2 units respectively, one can express this weight matrix as $W \approx UV^T$, where U and V are of sizes $m_1 \times k$ and $m_2 \times k$, respectively. Furthermore, it is assumed that $k \ll \min\{m_1, m_2\}$. This phenomenon occurs because of several peculiarities in the training process. For example, the features and weights in a neural network tend to *co-adapt* because of different parts of the network training at different rates. Therefore, the faster parts of the network often adapt to the slower parts. As a result, there is a lot of redundancy in the network both in terms of the features and the weights, and the full expressivity of the network is never utilized. In such a case, one can replace the pair of layers (containing weight matrix W) with three layers of size m_1, k, and m_2. The weight matrices between the first pair of layers is U and the weight matrix between the second pair of layers is V^T. Even though the new matrix is deeper, it is better regularized as long as $W - UV^T$ only contains noise. Furthermore, the matrices U and V require $(m_1 + m_2) \cdot k$ parameters, which is less than the number of parameters in W as long as k is less than half the harmonic mean of m_1 and m_2:

$$\frac{\text{Parameters in } W}{\text{Parameters in } U, V} = \frac{m_1 \cdot m_2}{k(m_1 + m_2)} = \frac{\text{HARMONIC-MEAN}(m_1, m_2)}{2k}$$

As shown in [96], more than 95% of the parameters in the neural network are redundant, and therefore a low value of the rank k suffices for approximation.

One can also fine-tune U and V with back-propagation. An important point is that fine-tuning U and V must be done *after* completion of the learning of W and initializing U and V so that $UV^T \approx W$. For example, if we replaced the pair of layers corresponding to W with the three layers containing the two weight matrices of the same shape as U and V^T and trained from scratch, good results may not be obtained. This is because co-adaptation will occur again during training, and the resulting matrices U and V will have a rank even lower than k. As a result, under-fitting might occur. Matrix factorization can also be applied after weight pruning.

Hash-Based Compression

One can reduce the number of parameters to be stored by forcing randomly chosen entries of the weight matrix to take on shared values of the parameters. The random choice is achieved with the application of a hash function on the entry position (i, j) in the matrix. For example, imagine a situation where we have a weight matrix of size 100×100 with 10^4 entries. In such a case, one can hash each weight to a value in the range $\{1, \ldots 1000\}$ to create 1000 groups. Each of these groups will contain an average of 10 connections that will share weights. Backpropagation can handle shared weights using the approach discussed in section 2.6.6. This approach requires a space requirement of only 1000 for the matrix, which is 10% of the original space requirement. Note that one could instead use a matrix of size 100×10 to achieve the same compression, but the key point is that using shared weights does not hurt the expressivity of the model as much as would reducing the size of the weight matrix *a priori*. More details of this approach are discussed in [68].

Leveraging Mimic Models

Some interesting results in [14, 55] show that it is possible to significantly compress a model by creating a new training data set from a trained model, which is easier to model. This "easier" training data can be used to train a much smaller network without significant loss of accuracy. This smaller model is referred to as a *mimic model*. The following steps are used to create the mimic model:

1. A model is created on the original training data. This model might be very large and would not be appropriate to use in space-constrained settings. It is assumed that the model outputs softmax probabilities of the different classes. This model is also referred to as the teacher model.

2. New training data is created by passing unlabeled examples through the trained network. The targets in the newly created training data are set to the softmax probability outputs of the trained model on the unlabeled examples. Since unlabeled data is often copious, it is possible to create a lot of training data in this way. The new training data contains soft (probabilistic) targets rather than discrete labels, the crucial effects of which will be discussed later.

3. A much smaller and shallower network, referred to as the mimic or *student* model, is trained using this synthesized training data. This model can be easily deployed in space-constrained settings. The accuracy of the mimic model often does not substantially degrade with respect to the original neural model.

A remarkable observation is that a shallow neural model might perform poorly with training on the original training data, whereas the mimic model of similar size performs very well. A number of possible reasons have been hypothesized for this phenomenon [14]:

1. Mislabeled examples in the original training data cause unnecessary complexity in the trained model. The new training data will often contain soft probabilities "disagreeing" with the original labels.

2. If there are complex regions of the decision space, the teacher model has the sophistication needed to simplify them in the most accurate way. Simplifying complexity selectively improves the performance of shallow models.

3. The original targets might depend on inputs that are not available in the training data. On the other hand, the teacher-created labels depend only on the available inputs. This makes the model simpler to learn and washes away unexplained complexity.

4. The original training data contains targets with 0/1 values, whereas the newly created training contains (more informative) soft targets. This is particularly useful in one-hot encoded multilabel targets with correlations across classes.

One can view some of the above benefits as a kind of regularization effect. The results in [14] are stimulating, because they show that deep networks are not *theoretically* necessary, although the regularization effect of depth is practically necessary when working with the original training data. The mimic model enjoys the benefits of this regularization effect by using the artificially created targets instead of depth.

4.10 Summary

This chapter discusses the problems of training deep neural networks. The vanishing and the exploding gradient problems are introduced along with the challenges associated with varying sensitivity of the loss function to different optimization variables. Certain types of activation functions like ReLU are less sensitive to this problem. However, the use of the ReLU can sometimes lead to dead neurons, if one is not careful about the learning rate. The type of gradient descent used to accelerate learning is also important for more efficient executions. Modified stochastic gradient-descent methods include the use of Nesterov momentum, AdaGrad, AdaDelta, RMSProp, and Adam. All these methods encourage gradient-steps that accelerate the learning process.

Numerous methods have been introduced for addressing the problem of cliffs with the use of second-order optimization methods. In particular, Hessian-free optimization is seen as an effective approach for handling many of the underlying optimization issues. An exciting method that has been used recently to improve learning rates is the use of batch normalization. Batch normalization transforms the data layer by layer in order to ensure that the scaling of different variables is done in an optimum way. The use of batch normalization has become extremely common in different types of deep networks. Numerous methods have been proposed for accelerating and compressing neural network algorithms. Acceleration is often achieved via hardware improvements, whereas compression is achieved with algorithmic tricks.

4.11 Bibliographic Notes and Software Resources

Nesterov's algorithm for gradient descent may be found in [364]. The delta-bar-delta method was proposed by [228]. The AdaGrad algorithm was proposed in [110]. The RMSProp algo-

rithm is discussed in [204]. Another adaptive algorithm using stochastic gradient descent, which is *AdaDelta*, is discussed in [578]. This algorithms shares some similarities with second-order methods, and in particular to the method in [447]. The Adam algorithm, which is a further enhancement along this line of ideas, is discussed in [251]. The practical importance of initialization and momentum in deep learning is discussed in [494]. Beyond the use of the stochastic gradient method, the use of coordinate descent has been proposed [279]. The strategy of *Polyak averaging* is discussed in [394].

Several of the challenges associated with the vanishing and exploding gradient problems are discussed in [147, 215, 381]. Ideas for parameter initialization that avoid some of these problems are discussed in [147]. The gradient clipping rule was discussed by Mikolov in his PhD thesis [336]. A discussion of the gradient clipping method in the context of recurrent neural networks is provided in [381]. The ReLU activation function was introduced in [174], and several of its interesting properties are explored in [148, 232].

A description of several second-order gradient optimization methods (such as the Newton method) is provided in [41, 309, 570]. The basic principles of the conjugate gradient method have been described in several classical books and papers [41, 199, 463], and the work in [324, 325] discusses applications to neural networks. The work in [327] leverages a Kronecker-factored curvature matrix for fast gradient descent. Another way of approximating the Newton method is the quasi-Newton method [279, 309], with the simplest approximation being a diagonal Hessian [25]. The acronym BFGS stands for the Broyden-Fletcher-Goldfarb-Shanno algorithm. A variant known as limited memory BFGS or L-BFGS [279, 309] does not require as much memory. Another popular second-order method is the Levenberg–Marquardt algorithm. This approach is, however, defined for squared loss functions and cannot be used with many forms of cross-entropy or log-losses that are common in neural networks. Overviews of the approach may be found in [139, 309]. General discussions of different types of nonlinear programming methods are provided in [24, 39].

The stability of neural networks to local minima is discussed in [91, 443]. Batch normalization methods were introduced recently in [225]. A method that uses whitening for batch normalization is discussed in [98], although the approach seems not to be practical. Batch normalization requires some minor adjustments for recurrent networks [84], although a more effective approach for recurrent networks is that of *layer normalization* [15]. In this method (cf. section 8.3.1), a single training case is used for normalizing all units in a layer, rather than using mini-batch normalization of a single unit. The approach is useful for recurrent networks. An analogous notion to batch normalization is that of weight normalization [436], in which the magnitudes and directions of the weight vectors are decoupled during the learning process. Related training tricks are discussed in [373].

A broader discussion of accelerating machine learning algorithms with GPUs may be found in [665]. Various types of parallelization tricks for GPUs are discussed in [77, 94, 262], and specific discussions on convolutional neural networks are provided in [563]. Model compression with regularization is discussed in [178, 179]. A related model compression method is proposed in [224]. The use of mimic models for compression is discussed in [14, 55]. A related approach is discussed in [212]. The leveraging of parameter redundancy for compressing neural networks is discussed in [96]. The compression of neural networks with the hashing trick is discussed in [68].

Software Resources

All the training algorithms discussed in this chapter, including batch normalization options, are supported by numerous deep learning frameworks like *Caffe* [596], *Torch* [597],

Theano [598], and *TensorFlow* [599]. Several software libraries are available for Bhyperparameter optimization, such as *Hyperopt* [637], *Spearmint* [639], and *SMAC* [638]. Pointers to the NVIDIA cuDNN and its supported deep learning frameworks discussed in [664, 666].

4.12 Exercises

1. Some networks, such as recurrent neural networks, use shared weights across multiple layers. Discuss why vanishing and exploding gradients are more likely in this case.

2. Consider a 2-feature linear regression problem in which x_1 always lies in the range $[8, 9]$ and x_2 always lies in the range $[8000, 9000]$. Comment on why gradient descent is likely to perform more efficiently after feature normalization by inspecting the objective function over a toy training set with three examples.

3. Suppose that you have the quadratic function $f(x) = ax^2 + bx + c$ with optimum value st $x = -b/2a$. Show that a single Newton step starting at any point $x = x_0$ will always lead to $x = -b/2a$ irrespective of the value of x_0. How does this result show that a Newton update to reach a maximum rather than a minimum?

4. Consider the objective function $f(x) = [x(x-2)]^2 + x^2$. Write its Newton update starting at $x = 1$. Discuss why line search is important in this case.

5. The Hessian H of a strongly convex quadratic function always satisfies $\overline{x}^T H \overline{x} > 0$ for any nonzero vector $\overline{x}$. For such problems, show that all conjugate directions are linearly independent.

6. Show that if the dot product of a d-dimensional vector $\overline{v}$ with d linearly independent vectors is 0, then $\overline{v}$ must be the zero vector.

7. The chapter uses steepest descent directions to iteratively generate conjugate directions. Suppose we pick d *arbitrary* directions $\overline{v}_0 \ldots \overline{v}_{d-1}$ that are linearly independent. Show that (with appropriate choice of β_{ti}) we can start with $\overline{q}_0 = \overline{v}_0$ and generate successive conjugate directions in the following form:

$$\overline{q}_{t+1} = \overline{v}_{t+1} + \sum_{i=0}^{t} \beta_{ti} \overline{q}_i$$

   Discuss why this approach is more expensive than the one discussed in the chapter.

8. The definition of β_t in section 4.7.1 ensures that $\overline{q}_t$ is conjugate to $\overline{q}_{t+1}$. This exercise systematically shows that *any* direction $\overline{q}_i$ for $i \leq t$ satisfies $\overline{q}_i^T H \overline{q}_{t+1} = 0$.
   [Hint: Prove (b), (c), and (d) *jointly* with induction on t while staring at (a).]

   (a) Recall from Equation 4.17 that $H\overline{q}_i = [\nabla L(\overline{W}_{i+1}) - \nabla L(\overline{W}_i)]/\delta_i$ for quadratic loss functions, where δ_i depends on ith step-size. Combine this condition with Equation 4.15 to show the following for all $i \leq t$:

$$\delta_i [\overline{q}_i^T H \overline{q}_{t+1}] = -[\nabla L(\overline{W}_{i+1}) - \nabla L(\overline{W}_i)]^T [\nabla L(\overline{W}_{t+1})] + \delta_i \beta_t (\overline{q}_i^T H \overline{q}_t)$$

   Also show that $[\nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)] \cdot \overline{q}_i = \delta_t \overline{q}_i^T H \overline{q}_t$.

(b) Show that $\nabla L(\overline{W}_{t+1})$ is orthogonal to each $\overline{q}_i$ for $i \leq t$. [The proof for the case when $i = t$ is trivial because the gradient at line-search termination is always orthogonal to the search direction.]

(c) Show that the loss gradients at $\overline{W}_0 \ldots \overline{W}_{t+1}$ are mutually orthogonal.

(d) Show that $\overline{q}_i^T H \overline{q}_{t+1} = 0$ for $i \leq t$. [The case for $i = t$ is trivial.]

9. Revisit Exercise 11 of Chapter 2 and discuss why the tanh activation is less likely to cause gradient descent to zigzag than sigmoid activation. Also discuss why the vanishing gradient is less likely with tanh activation.

10. Weight pruning, matrix factorization, and mimic models are all compression techniques. If you had to use them in combination, then in what order would you apply them and why?

Chapter 5

Teaching Deep Learners to Generalize

"All generalizations are dangerous, even this one."–Alexandre Dumas

5.1 Introduction

Neural networks are powerful learners that have repeatedly proven to be capable of learning complex functions in many domains. However, the great power of neural networks is also their greatest weakness; neural networks often simply overfit the training data if care is not taken to design the learning process carefully. In practical terms, what overfitting means is that a neural network will provide excellent prediction performance on the training data that it is built on, but will perform poorly on unseen test instances. This is caused by the fact that the learning process often remembers random artifacts of the training data that do not generalize well to the test data. Extreme forms of overfitting are referred to as *memorization*. A helpful analogy is to think of a child who can solve all the analytical problems for which he or she has seen the solutions, but is unable to provide useful solutions to a new problem. However, if the child is exposed to the solutions of more and more different types of problems, he or she will be more likely to solve a new problem by abstracting out the essence of the patterns that are repeated across different problems and their solutions. Machine learning proceeds in a similar way by identifying patterns that are useful for prediction. For example, in a spam detection application, if the pattern "*Free Money!!*" occurs thousands of times in spam emails, the machine learner generalizes this rule to identify spam email instances it has not seen before. On the other hand, a prediction that is based on the patterns seen in a tiny training data set of two emails will lead to good performance on the two emails present in the training data but (usually) not on new emails. The ability of a learner to provide useful predictions for instances it has not seen before is referred to as *generalization*.

Generalization is a useful practical property, and is therefore the holy grail in all machine learning applications. After all, if the training examples are already labeled, there is no practical use of predicting such examples again. For example, in an image-captioning

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_5

application, one is always looking to use the labeled images in order to learn captions for images that the learner has not seen before.

5.1.1 Example: Linear Regression

In order to understand the problem of generalization, consider a simple single-layer neural network on a data set with five attributes, where we use the identity activation to learn a real-valued target variable. This architecture is almost identical to that of Figure 1.3, except that the identity activation function is used in order to predict a real-valued target. Therefore, the network tries to learn the following function:

$$\hat{y} = \sum_{i=1}^{5} w_i \cdot x_i \tag{5.1}$$

Consider a situation in which the observed target value is real and is always twice the value of the first attribute, whereas other attributes are completely unrelated to the target. However, we have only four training instances, which is one less than the number of features (free parameters). For example, the training instances could be as follows:

x_1	x_2	x_3	x_4	x_5	y
1	1	0	0	0	2
2	0	1	0	0	4
3	0	0	1	0	6
4	0	0	0	1	8

The correct parameter vector in this case is $\overline{W} = [2, 0, 0, 0, 0]$ based on the known relationship between the first feature and target. The training data also provides zero error with this solution, although the relationship needs to be *learned* from the given instances since it is not given to us a priori. However, the problem is that the number of training points is fewer than the number of parameters and it is possible to find an infinite number of solutions with zero error. For example, the parameter set $[0, 2, 4, 6, 8]$ also provides zero error *on the training data*. However, if we used this solution on unseen test data, it is likely to provide very poor performance because the learned parameters are spuriously inferred and are unlikely to *generalize* well to new points in which the target is twice the first attribute (and other attributes are random). This type of spurious inference is caused by the paucity of training data, where random nuances are encoded into the model. As a result, the solution does not generalize well to unseen test data. This situation is almost similar to learning by rote, which is highly predictive for training data but not predictive for unseen test data. Increasing the number of training instances improves the generalization power of the model, whereas increasing the complexity of the model reduces its generalization power. At the same time, when a lot of training data is available, an overly simple model is unlikely to capture complex relationships between the features and target. A good rule of thumb is that the total number of training data points should be at least 2 to 3 times larger than the number of parameters in the neural network, although the precise number of data instances depends on the specific model at hand. In general, models with a larger number of parameters are said to have *high capacity*, and they require a larger amount of data in order to gain generalization power to unseen test data.

For example, if we change the training data in the table above to a different set of four points, we are likely to learn a completely different set of parameters (from the random

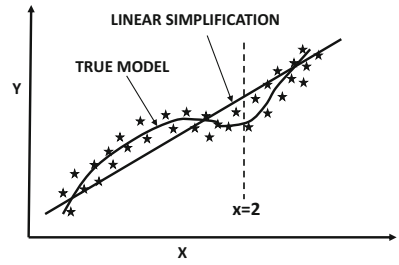


Figure 5.1: An example of a nonlinear distribution in which one would expect a model with $d = 3$ to work better than a linear model with $d = 1$.

nuances of those points). This new model is likely to yield a completely different prediction *on the same test instance* as compared to the predictions using the first training data set. Therefore, models that overfit the training data are said of have high *variance*.

5.1.2 Example: Polynomial Regression

Imagine a situation in which we attempt to predict the variable y from x using the following formula for polynomial regression:

$$\hat{y} = \sum_{i=0}^{d} w_i x^i \tag{5.2}$$

This is a model that uses $(d + 1)$ parameters $w_0 \ldots w_d$ in order to explain pairs (x, y) available to us. One could implement this model by using a neural network with d inputs corresponding to x, $x^2 \ldots x^d$, and a single bias neuron whose coefficient is w_0. The loss function uses the squared difference between the observed value y and predicted value $\hat{y}$. In general, larger values of d can capture better nonlinearity. For example, in the case of Figure 5.1, a nonlinear model with $d = 4$ should be able to fit the data better than a linear model with $d = 1$, *given an infinite amount (or a lot) of data*. However, when working with a small, finite data set, this does not always turn out to be the case.

If we have $(d + 1)$ or less training pairs (x, y), it is possible to fit the data exactly with zero error *irrespective of how poorly these training pairs reflect the true distribution*. For example, consider a situation in which we have five particularly bad (unrepresentative) training points available. One can show that it is possible to fit the training points exactly with zero error using a polynomial of degree 4. This does not, however, mean that zero error will be achieved on unseen test data. An example of this situation is illustrated in Figure 5.2, where both the linear and polynomial models on three sets of five randomly chosen data points are shown. It is clear that the linear model is stable, although it is unable to exactly model the curved nature of the true data distribution. On the other hand, even though the polynomial model is capable of modeling the true data distribution more closely, it varies wildly over the different training data sets. Therefore, the same test instance at $x = 2$ (shown in Figure 5.2) would receive similar predictions from the linear model, but would receive very different predictions from the polynomial model over different choices of training data sets. The behavior of the polynomial model is, of course, undesirable from a practitioner's point of view, who would expect similar predictions for a particular test instance, even when different samples of the training data set are used. Since all the different predictions of the polynomial model cannot be correct, it is evident that the increased power of the

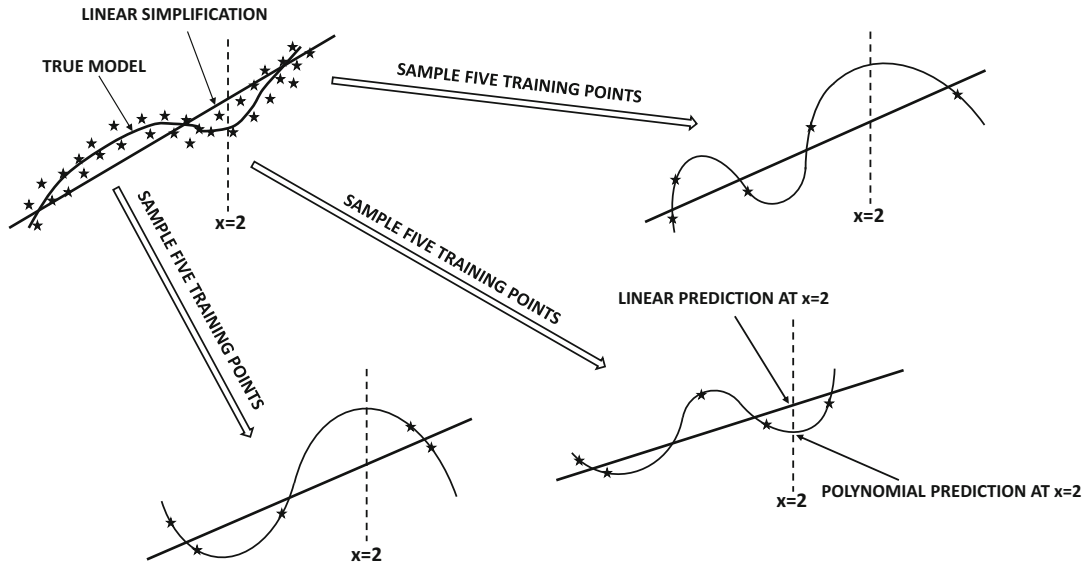


Figure 5.2: **Overfitting with increased model complexity:** The linear model does not change much with the training data, whereas the polynomial model changes drastically. As a result, the inconsistent predictions of the polynomial model at $x = 2$ are often more inaccurate than those of the linear model. The polynomial model does have the ability to outperform the linear model *if enough training data is provided*.

polynomial model over the linear model actually increases the error rather than reducing it. This difference in predictions for the same test instance (but different training data sets) is manifested as the *variance* of a model. As evident from Figure 5.2, models with high variance tend to memorize random artifacts of the training data, causing inconsistency and inaccuracy in the prediction of unseen test instances. It is noteworthy that a polynomial model with higher degree is inherently more powerful than a linear model because the higher-order coefficients could always be set to 0; however, it is unable to achieve its full potential when the amount of data is limited. Simply speaking, the variance inherent in the finiteness of the data set causes increased complexity to be counterproductive. This trade-off between the power of a model and its performance on limited data is captured with the *bias-variance trade-off*.

There are several tell-tale signs of overfitting:

1. When a model is trained on different data sets, the same test instance might obtain very different predictions. This is a sign that the training process is memorizing the nuances of the specific training data set, rather than learning patterns that generalize to unseen test instances. Note that the three predictions at $x = 2$ in Figure 5.2 are quite different for the polynomial model. This is not quite the case for the linear model.

2. The gap between the error of predicting training instances and unseen test instances is rather large. Note that in Figure 5.2, the predictions at the unseen test point $x = 2$ are often more inaccurate in the polynomial model than in the linear model. On the other hand, the training error is always zero for the polynomial model, whereas the training error is usually nonzero for the linear model.

Because of the large gaps between training and test error, models are often tested on unseen portions of the training data. These unseen portions of the training data are often held out early on, and then used in order to make different types of algorithmic decisions such as parameter tuning. This set of points is referred to as the *validation set*. The final accuracy is tested on a fully out-of-sample set of points that was not used for either model building or for parameter tuning. The error on out-of-sample test data is also referred to as the *generalization error*.

Neural networks are large, and they might have millions of parameters in complex applications. In spite of these challenges, there are a number of tricks that one can use in order to ensure that overfitting is not a problem. The key methods for avoiding overfitting in a neural network are as follows:

1. *Penalty-based regularization:* Penalty-based regularization is the most common technique used by neural networks in order to avoid overfitting. The idea in regularization is to create a penalty or other types of constraints on the parameters in order to favor simpler models. For example, in the case of polynomial regression, a possible constraint on the parameters would be to ensure that at most k different values of w_i are non-zero. This will ensure simpler models. However, since it is hard to impose such constraints explicitly, a simpler approach is to impose a softer penalty like $\lambda \sum_{i=0}^{d} w_i^2$ and add it to the loss function. Such an approach roughly amounts to multiplying each parameter w_i with a multiplicative decay factor of $(1 - \alpha\lambda)$ during each update at learning rate α.

2. *Generic and tailored ensemble methods:* Many ensemble methods are not specific to neural networks. We will discuss bagging and subsampling, which are two of the simplest ensemble methods that can be implemented for virtually any model or learning problem. There are also several ensemble methods that are specifically designed for neural networks. A straightforward approach is to average the predictions of different neural architectures obtained by quick and dirty hyper-parameter optimization. *Dropout* is another ensemble technique that is designed for neural networks. This chapter will discuss some of these methods.

3. *Early stopping:* In early stopping, the iterative optimization method is terminated early without converging to the optimal solution on the training data. The stopping point is determined using a portion of the training data that is not used for model building. One terminates when the error on the held-out data begins to rise. Even though this approach is not optimal for the training data, it seems to perform well on the test data because the stopping point is determined using held-out data.

4. *Pretraining:* Pretraining is a form of learning in which a greedy algorithm is used to find a good initialization. The weights in different layers of the neural network are trained sequentially in greedy fashion. These trained weights are used as a good starting point for the overall process of learning. Pretraining can be shown to be an indirect form of regularization.

5. *Continuation and curriculum methods:* These methods perform more effectively by first training simple models, and then making them more complex. The idea is that it is easy to train simpler models without overfitting. Furthermore, starting with the optimum point of the simpler model provides a good initialization for a complex model that is closely related to the simpler model. It is noteworthy that some of these methods can be considered similar to pretraining, which transforms solutions from

the simple to the complex by decomposing a deep neural network into a set of shallow layers for training.

6. *Sharing parameters with domain-specific insights:* In some data-domains like text and images, one often has some insight about the structure of the parameter space. In such cases, some of the parameters in different parts of the network can be set to the same value. This reduces the number of degrees of freedom of the model. Such an approach is used in recurrent neural networks (for sequence data) and convolutional neural networks (for image data).

This chapter will first discuss the issue of model generalization in a generic way by introducing some theoretical results associated with the bias-variance trade-off. Subsequently, the different ways of reducing overfitting will be discussed.

An interesting observation is that several forms of regularization can be shown to be roughly equivalent to the injection of noise in either the input data or the hidden variables. For example, it can be shown that many penalty-based regularizers are equivalent to the addition of noise [43]. Furthermore, even the use of *stochastic* gradient descent instead of gradient descent can be viewed as a kind of noise addition to the steps of the algorithm. As a result, stochastic gradient descent often shows good accuracy on the test data, even though its performance on the training data might not be as good as that of gradient descent. Furthermore, some ensemble techniques like *Dropout* and data perturbation are equivalent to injecting noise. Throughout this chapter, the similarities between noise injection and regularization will be discussed where needed.

Even though a natural way of avoiding overfitting is to simply build smaller networks (with fewer units and parameters), it has often been observed that it is better to build large networks and then regularize them in order to avoid overfitting. This is because large networks retain the *option* of building a more complex model if it is truly warranted. At the same time, the regularization process can smooth out the random artifacts that are not supported by sufficient data. By using this approach, we are giving the model the choice to decide what complexity it needs, rather than making a rigid decision for the model up front (which might even underfit the data).

Supervised settings tend to be more prone to overfitting than unsupervised settings, and supervised problems are therefore the main focus of the literature on generalization. To understand this point, consider that a supervised application tries to learn a single target variable and might have hundreds of input (explanatory) variables. It is easy to overfit the process of learning a very focused goal because a limited degree of supervision (e.g., binary label) is available for each training example. On the other hand, an unsupervised application has the same number of target variables as the explanatory variables. In the latter case, overfitting is less likely because a single training example has a larger number of bits of information. Nevertheless, regularization is still used in unsupervised applications, especially when the intent is to impose a desired structure on the learned representations.

Chapter Organization

This chapter is organized as follows. The next section introduces the bias-variance trade-off. The practical implications of the bias-variance trade-off for model training are discussed in section 5.3. The use of penalty-based regularization to reduce overfitting is presented in section 5.4. Ensemble methods are introduced in section 5.5. Early stopping methods are discussed in section 5.6. Methods for unsupervised pretraining are discussed in section 5.7.

Continuation and curriculum learning methods are presented in section 5.8. Parameter sharing methods are discussed in section 5.9. Unsupervised forms of regularization are discussed in section 5.10. A summary is given in section 5.11.

5.2 The Bias-Variance Trade-Off

The introduction section provides an example of how a polynomial model fits a smaller training data set, leading to the predictions on unseen test data being more erroneous than are the predictions of a (simpler) linear model. This is because a polynomial model requires more data in order to not be misled by random artifacts of the training data set. The fact that more powerful models do not always win in terms of prediction accuracy with a finite data set is the key take-away from the bias-variance trade-off.

The bias-variance trade-off states that the squared error of a learning algorithm can be partitioned into three components:

1. *Bias:* The bias is the error caused by the simplifying assumptions in the model, which causes certain test instances to have consistent errors across different choices of training data sets. Even if the model has access to an infinite source of training data, the bias cannot be removed. For example, in the case of Figure 5.2, the linear model has a higher model bias than the polynomial model, because it can never fit the (slightly curved) data distribution exactly, no matter how much data is available. The prediction of a particular out-of-sample test instance at $x = 2$ will always have an error in a particular direction when using a linear model for any choice of training sample. If we assume that the linear and curved lines in the top left of Figure 5.2 were estimated using an infinite amount of data, then the difference between the two at any particular values of x is the bias. An example of the bias at $x = 2$ is shown in Figure 5.2.

2. *Variance:* Variance is caused by the inability to learn all the parameters of the model in a statistically robust way, especially when the data is limited and the model tends to have a larger number of parameters. The presence of higher variance is manifested by overfitting to the specific training data set at hand. Therefore, if different choices of training data sets are used, different predictions will be provided for the same test instance. Note that the linear prediction provides similar predictions at $x = 2$ in Figure 5.2, whereas the predictions of the polynomial model vary widely over different choices of training instances. In many cases, the widely inconsistent predictions at $x = 2$ are wildly incorrect predictions, which is a manifestation of model variance. Therefore, the polynomial predictor has a higher variance than the linear predictor in Figure 5.2.

3. *Noise:* The noise is caused by the inherent error in the data. For example, all data points in the scatter plot vary from the true model in the upper-left corner of Figure 5.2. If there had been no noise, all points in the scatter plot would overlap with the curved line representing the true model.

The above description provides a qualitative view of the bias-variance trade-off. In the following, we will provide a more formal and mathematical view.

Formal View

We assume that the base distribution from which the training data set is generated is denoted by $\mathcal{B}$. One can generate a data set $\mathcal{D}$ from this base distribution:

$$\mathcal{D} \sim \mathcal{B} \tag{5.3}$$

One could draw the training data in many different ways, such as selecting only data sets of a particular size. For now, assume that we have some well-defined generative process according to which training data sets are drawn from $\mathcal{B}$. The analysis below does not rely on the specific mechanism with which training data sets are drawn from $\mathcal{B}$.

Access to the base distribution $\mathcal{B}$ is equivalent to having access to an infinite resource of training data, because one can use the base distribution an unlimited number of times to generate training data sets. In practice, such base distributions (i.e., infinite resources of data) are not available. As a practical matter, an analyst uses some data collection mechanism to collect only *one finite instance* of $\mathcal{D}$. However, the conceptual existence of a base distribution from which other training data sets can be generated is useful in theoretically quantifying the sources of error in training on this finite data set.

Now imagine that the analyst had a set of t test instances in d dimensions, denoted by $\overline{Z}_1 \ldots \overline{Z}_t$. The dependent variables of these test instances are denoted by $y_1 \ldots y_t$. For clarity in discussion, let us assume that the test instances and their dependent variables were also generated from the same base distribution $\mathcal{B}$ by a third party, but the analyst was provided access only to the feature representations $\overline{Z}_1 \ldots \overline{Z}_t$, and no access to the dependent variables $y_1 \ldots y_t$. Therefore, the analyst is tasked with job of using the single finite instance of the training data set $\mathcal{D}$ in order to predict the dependent variables of $\overline{Z}_1 \ldots \overline{Z}_t$.

Now assume that the relationship between the dependent variable y_i and its feature representation $\overline{Z}_i$ is defined by the *unknown* function $f(\cdot)$ as follows:

$$y_i = f(\overline{Z}_i) + \epsilon_i \tag{5.4}$$

Here, the notation ϵ_i denotes the intrinsic noise, which is independent of the model being used. The value of ϵ_i might be positive or negative, although it is assumed that $E[\epsilon_i] = 0$. If the analyst knew what the function $f(\cdot)$ corresponding to this relationship was, then they could simply apply the function to each test point $\overline{Z}_i$ in order to approximate the dependent variable y_i, with the only remaining uncertainty being caused by the intrinsic noise.

The problem is that the analyst does not know what the function $f(\cdot)$ is in practice. Note that this function is used within the generative process of the base distribution $\mathcal{B}$, and the entire generating process is like an oracle that is unavailable to the analyst. The analyst only has examples of the input and output of this function. Clearly, the analyst would need to develop some type of *model* $g(\overline{Z}_i, \mathcal{D})$ using the training data in order to *approximate* this function in a data-driven way.

$$\hat{y}_i = g(\overline{Z}_i, \mathcal{D}) \tag{5.5}$$

Note the use of the circumflex (i.e., the symbol '^') on the variable $\hat{y}_i$ to indicate that it is a *predicted* value by a specific algorithm rather than the observed (true) value of y_i.

All prediction functions of learning models (including neural networks) are examples of the estimated function $g(\cdot, \cdot)$. Some algorithms (such as linear regression and perceptrons)

can even be expressed in a concise and understandable way:

$$g(\overline{Z}_i, \mathcal{D}) = \underbrace{\overline{W} \cdot \overline{Z}_i}_{\text{Learn } \overline{W} \text{ with } \mathcal{D}} \quad \text{[Linear Regression]}$$

$$g(\overline{Z}_i, \mathcal{D}) = \underbrace{\text{sign}\{\overline{W} \cdot \overline{Z}_i\}}_{\text{Learn } \overline{W} \text{ with } \mathcal{D}} \quad \text{[Perceptron]}$$

Most neural networks are expressed algorithmically as compositions of multiple functions computed at different nodes. The choice of computational function includes the effect of its specific parameter setting, such as the coefficient vector $\overline{W}$ in a perceptron. Neural networks with a larger number of units will require more parameters to fully learn the function. This is where the variance in predictions arises on the same test instance; a model with a large parameter set $\overline{W}$ will learn very different values of these parameters, when a different choice of the training data set is used. Consequently, the prediction of the same test instance will also be very different for different training data sets. These inconsistencies add to the error, as illustrated in Figure 5.2.

The goal of the bias-variance trade-off is to quantify the expected error of the learning algorithm in terms of its bias, variance, and the (data-specific) noise. For generality in discussion, we assume a numeric form of the target variable, so that the error can be intuitively quantified by the *mean-squared error* between the predicted values $\hat{y}_i$ and the observed values y_i. This is a natural form of error quantification in regression, although one can also use it in classification in terms of probabilistic predictions of test instances. The mean squared error, MSE, of the learning algorithm $g(\cdot, \mathcal{D})$ is defined over the set of test instances $\overline{Z}_1 \ldots \overline{Z}_t$ as follows:

$$MSE = \frac{1}{t} \sum_{i=1}^{t} (\hat{y}_i - y_i)^2 = \frac{1}{t} \sum_{i=1}^{t} (g(\overline{Z}_i, \mathcal{D}) - f(\overline{Z}_i) - \epsilon_i)^2$$

The best way to estimate the error in a way that is independent of the specific choice of training data set is to compute the *expected* error over different choices of training data sets:

$$\begin{aligned} E[MSE] &= \frac{1}{t} \sum_{i=1}^{t} E[(g(\overline{Z}_i, \mathcal{D}) - f(\overline{Z}_i) - \epsilon_i)^2] \\ &= \frac{1}{t} \sum_{i=1}^{t} E[(g(\overline{Z}_i, \mathcal{D}) - f(\overline{Z}_i))]^2 + \frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t} \end{aligned}$$

The second relationship is obtained by expanding the quadratic expression on the right-hand side of the first equation, and then using the fact that the average value of ϵ_i over a large number of test instances is 0.

The right-hand side of the above expression can be further decomposed by adding and subtracting $E[g(\overline{Z}_i, \mathcal{D})]$ within the squared term on the right-hand side:

$$E[MSE] = \frac{1}{t} \sum_{i=1}^{t} E[\{(f(\overline{Z}_i) - E[g(\overline{Z}_i, \mathcal{D})]) + (E[g(\overline{Z}_i, \mathcal{D})] - g(\overline{Z}_i, \mathcal{D}))\}^2] + \frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t}$$

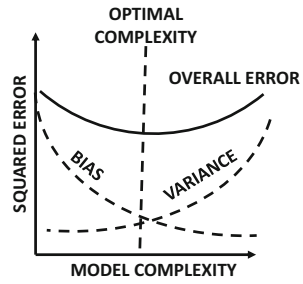


Figure 5.3: The trade-off between bias and variance usually causes a point of optimal model complexity.

One can expand the quadratic polynomial on the right-hand side to obtain the following:

$$\begin{aligned} E[MSE] =& \frac{1}{t}\sum_{i=1}^{t} E[\{f(\overline{Z}_i) - E[g(\overline{Z}_i, \mathcal{D})]\}^2] \\ &+ \frac{2}{t}\sum_{i=1}^{t} \{f(\overline{Z}_i) - E[g(\overline{Z}_i, \mathcal{D})]\}\{E[g(\overline{Z}_i, \mathcal{D})] - E[g(\overline{Z}_i, \mathcal{D})]\} \\ &+ \frac{1}{t}\sum_{i=1}^{t} E[\{E[g(\overline{Z}_i, \mathcal{D})] - g(\overline{Z}_i, \mathcal{D})\}^2] + \frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t} \end{aligned}$$

The second term on the right-hand side of the aforementioned expression evaluates to 0 because one of the multiplicative factors is $E[g(\overline{Z}_i, \mathcal{D})] - E[g(\overline{Z}_i, \mathcal{D})]$. On simplification, we obtain the following:

$$E[MSE] = \underbrace{\frac{1}{t}\sum_{i=1}^{t} \{f(\overline{Z}_i) - E[g(\overline{Z}_i, \mathcal{D})]\}^2}_{\text{Bias}^2} + \underbrace{\frac{1}{t}\sum_{i=1}^{t} E[\{g(\overline{Z}_i, \mathcal{D}) - E[g(\overline{Z}_i, \mathcal{D})]\}^2]}_{\text{Variance}} + \underbrace{\frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t}}_{\text{Noise}}$$

In other words, the squared error can be decomposed into the (squared) bias, variance, and noise. The variance is the key term that prevents neural networks from generalizing. In general, the variance will be higher for neural networks that have a large number of parameters. On the other hand, too few model parameters can cause bias because there are not sufficient degrees of freedom to model the complexities of the data distribution. This trade-off between bias and variance with increasing model complexity is illustrated in Figure 5.3. Clearly, there is a point of optimal model complexity where the performance is optimized. Furthermore, paucity of training data will increase variance. However, careful choice of design can reduce overfitting. This chapter will discuss several such choices.

5.3 Generalization Issues in Model Tuning and Evaluation

There are several practical issues in the training of neural network models that one must be careful of because of the bias-variance trade-off. The first of these issues is associated with model tuning and hyperparameter choice. For example, if one tuned the neural network with

the same data that were used to train it, one would not obtain very good results because of overfitting. Therefore, the hyperparameters (e.g., regularization parameter) are tuned on a separate held-out set than the one on which the weight parameters on the neural network are learned.

Given a labeled data set, one needs to use this resource for training, tuning, and testing the accuracy of the model. Clearly, one cannot use the entire resource of labeled data for model building (i.e., learning the weight parameters). For example, using the same data set for both model building and testing grossly overestimates the accuracy. This is because the main goal of classification is to *generalize* a model of labeled data to unseen test instances. Furthermore, the portion of the data set used for *model selection* and *parameter tuning* also needs to be different from that used for model building. A common mistake is to use the same data set for both parameter tuning and final evaluation (testing). Such an approach partially mixes the training and test data, and the resulting accuracy is overly optimistic. A given data set should always be divided into three parts defined according to the way in which the data are used:

1. *Training data:* This part of the data is used to build the training model (i.e., during the process of learning the weights of the neural network). Multiple models may be constructed using different hyperparameters and architectures. This process sets the stage for *model selection*, in which the best algorithm is selected out of these different models. However, the actual *evaluation* of these algorithms for selecting the best model is not done on the training data, but on a separate validation data set to avoid favoring overfitted models.

2. *Validation data:* This part of the data is used for model selection and parameter tuning. The accuracy of various models constructed on the training data is tested using the validation set. As discussed in section 2.7.1 of Chapter 2, different combinations of parameters are sampled within a range and tested for accuracy on the validation set. The best combination of parameters is determined by using this accuracy. In a sense, validation data should be viewed as a kind of test data set to tune the parameters of the algorithm (e.g., learning rate, number of layers or units in each layer), or to select the best design choice (e.g., sigmoid versus tanh activation).

3. *Testing data:* This part of the data is used to test the accuracy of the final (tuned) model. It is important that the testing data are not even looked at during the process of parameter tuning and model selection to prevent overfitting. The testing data are *used only once at the very end of the process.* Furthermore, if the analyst uses the results on the test data to adjust the model in some way, then the results will be contaminated with knowledge from the testing data. The idea that one is allowed to look at a test data set only once is an extraordinarily strict requirement (and an important one). Yet, it is frequently violated in real-life benchmarks. The temptation to use what one has learned from the final accuracy evaluation is simply too high.

The division of the labeled data set into training data, validation data, and test data is shown in Figure 5.4. Strictly speaking, the validation data is also a part of the training data, because it influences the final model (although only the model building portion is often referred to as the training data). The division in the ratio of 2:1:1 is a conventional rule of thumb that has been followed since the nineties. However, it should not be viewed as a strict rule. For very large labeled data sets, one needs only a modest number of examples to estimate accuracy. When a very large data set is available, it makes sense to use as much

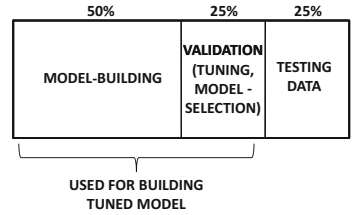


Figure 5.4: Partitioning a labeled data set for evaluation design

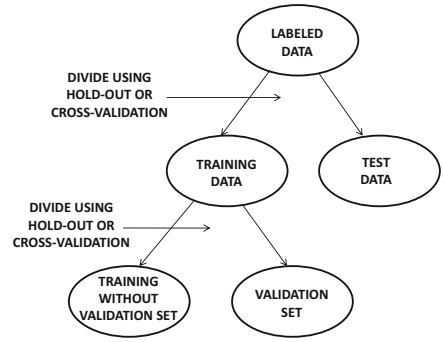


Figure 5.5: Hierarchical division into training, validation, and testing portions

of it for model building as possible, because the variance induced by the validation and evaluation stage is often quite low. A constant number of examples (e.g., less than a few thousand) in the validation and test data sets are sufficient to provide accurate estimates. The 2:1:1 division is a rule of thumb that has now become defunct, because it is inherited from an era in which data sets were small. In the modern era, where data sets are large, almost all of the points are used for training, and a modest (constant) number are used for validation and testing. It is not uncommon to have divisions such as 98:1:1.

5.3.1 Evaluating with Hold-Out and Cross-Validation

The aforementioned description of partitioning the labeled data into three segments is an implicit description of a method referred to as *hold-out* for segmenting the labeled data into various portions. However, the division into *three* parts is not done in one shot. Rather, the training data is first divided into *two* parts for training and testing. The testing part is then carefully hidden away from any further analysis *until the very end where it can be used only once*. The remainder of the data set is then divided again into the training and validation portions. This type of recursive division is shown in Figure 5.5.

The types of division at both levels of the hierarchy are conceptually identical. In the following, we will consistently use the terminology of the first level of division in Figure 5.5 into "training" and "testing" data, even though the same approach can also be used for the second-level division into model building and validation portions. This allows us to provide a unified description of evaluation processes at both levels of the division.

Hold-Out

In the hold-out method, a fraction of the instances are used to build the training model. The remaining instances, which are also referred to as the *held-out* instances, are used for testing. The accuracy of predicting the labels of the held-out instances is then reported as the overall accuracy. Such an approach ensures that the reported accuracy is not a result of overfitting to the specific data set, because different instances are used for training and testing. The approach, however, underestimates the true accuracy. Consider the case where the held-out examples have a higher presence of a particular class than the labeled data set. This means that the held-in examples have a lower average presence of the same class, which will cause a mismatch between the training and test data. Furthermore, the class-wise frequency of the held-in examples will always be inversely related to that of the held-out examples. This will lead to a consistent pessimistic bias in the evaluation. In spite of these weaknesses, the hold-out method has the advantage of being simple and efficient, which makes it a popular choice in large-scale settings. From a deep-learning perspective, this is an important observation because large data sets are common.

Cross-Validation

In the cross-validation method, the labeled data is divided into q equal segments. One of the q segments is used for testing, and the remaining $(q-1)$ segments are used for training. This process is repeated q times by using each of the q segments as the test set. The average accuracy over the q different test sets is reported. Note that this approach can closely estimate the true accuracy when the value of q is large. A special case is one where q is chosen to be equal to the number of labeled data points and therefore a single point is used for testing. Since this single point is left out from the training data, this approach is referred to as *leave-one-out cross-validation*. Although such an approach can closely approximate the accuracy, it is usually too expensive to train the model a large number of times. In fact, cross-validation is sparingly used in neural networks because of efficiency issues.

5.3.2 Issues with Training at Scale

One practical issue that arises in the specific case of neural networks is when the sizes of the training data sets are large. Therefore, while methods like cross-validation are well established to be superior choices to hold-out in traditional machine learning, their technical soundness is often sacrificed in favor of efficiency. In general, training time is such an important consideration in neural network modeling that many compromises have to be made to enable practical implementation.

A computational problem often arises in the context of grid search of hyperparameters (cf. section 2.7.1 of Chapter 2). Even a single hyperparameter choice can sometimes require a few days to evaluate, and a grid search requires the testing of a large number of possibilities. Therefore, a common strategy is to run the training process of each setting for a fixed number of epochs. Multiple runs are executed over different choices of hyperparameters in different threads of execution. Those choices of hyperparameters in which good progress is not made after a fixed number of epochs are terminated. In the end, only a few ensemble members are allowed to run to completion. One reason that such an approach works well is because the vast majority of the progress is often made in the early phases of the training. This process is also described in section 2.7.1 of Chapter 2.

5.3.3 How to Detect Need to Collect More Data

The high generalization error in a neural network may be caused by several reasons. First, the data itself might have a lot of noise, in which case there is little one can do in order to improve accuracy. Second, neural networks are hard to train, and the large error might be caused by the poor convergence behavior of the algorithm. The error might also be caused by high bias, which is referred to as *underfitting*. Finally, overfitting (i.e., high variance) may cause a large part of the generalization error. In most cases, the error is a combination of more than one of these different factors. However, one can detect overfitting in a specific training data set by examining the gap between the training and test accuracy. Overfitting is manifested *by a large gap between training and test accuracy*. It is not uncommon to have close to 100% training accuracy on a small training set, even when the test error is quite low. The first solution to this problem is to collect more data. With increased training data, the training accuracy will reduce, whereas the test/validation accuracy will increase. However, if more data is not available, one would need to use other techniques such as regularization in order to improve generalization performance.

5.4 Penalty-Based Regularization

Penalty-based regularization is the most common approach for reducing overfitting. In order to understand this point, let us revisit the example of the polynomial with degree d. In this case, the prediction $\hat{y}$ for a given value of x is as follows:

$$\hat{y} = \sum_{i=0}^{d} w_i x^i \tag{5.6}$$

It is possible to use a single-layer network with d inputs and a single bias neuron with weight w_0 in order to model this prediction. The ith input is x^i. This neural network uses linear activations, and the squared loss function for a set of training instances (x, y) from data set $\mathcal{D}$ can be defined as follows:

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2$$

As discussed in the example of Figure 5.2, a large value of d tends to increase overfitting. One possible solution to this problem is to reduce the value of d. In other words, using a model with *economy in parameters* leads to a simpler model. For example, reducing d to 1 creates a linear model that has fewer degrees of freedom and tends to fit the data in a similar way over different training samples. However, doing so does lose some expressivity when the data patterns are indeed complex. In other words, oversimplification reduces the expressive power of a neural network, so that it is unable to adjust sufficiently to the needs of different types of data sets.

How can one retain some of this expressiveness without causing too much overfitting? Instead of reducing the number of parameters in a hard way, one can use a *soft* penalty on the use of parameters. Furthermore, large (absolute) values of the parameters are penalized more than small values, because small values do not affect the prediction significantly. What kind of penalty can one use? The most common choice is L_2-regularization, which is also referred to as *Tikhonov regularization*. In such a case, the additional penalty is defined by the sum of squares of the values of the parameters. Then, for the regularization parameter $\lambda > 0$, one can define the regularized objective function J (including loss L) as follows:

$$J = \underbrace{\sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2}_{L} + \lambda \cdot \sum_{i=0}^{d} w_i^2$$

Increasing or decreasing the value of λ reduces the softness of the penalty. One can tune this parameter for optimum performance on a held-out portion of the data set. Using this type of approach provides greater flexibility than fixing the economy of the model up front. Consider the case of polynomial regression discussed above. Restricting the number of parameters up front severely constrains the learned polynomial to a specific shape (e.g., a linear model), whereas a soft penalty is able to control the shape of the learned polynomial in a more data-driven manner. In general, it has been experimentally observed that it is more desirable to use complex models (e.g., larger neural networks) with regularization rather than simple models without regularization. The former also provides greater flexibility by providing a tunable knob (i.e., regularization parameter), which can be chosen in a data-driven manner, rather than making a rigid decision on the size of the model up front.

How does regularization affect the updates in a neural network? For any given weight w_i in the neural network, the updates are defined by gradient descent (or the batched version of it):

$$w_i \Leftarrow w_i - \alpha \frac{\partial J}{\partial w_i}$$

Here, α is the learning rate. The use of L_2-regularization is roughly equivalent to the use of decay imposition after each parameter update:

$$w_i \Leftarrow w_i(1 - \alpha\lambda) - \alpha \frac{\partial L}{\partial w_i}$$

Note that the update above first multiplies the weight with the decay factor $(1 - \alpha\lambda)$, and then uses the gradient-based update. The decay of the weights can also be understood in terms of a biological interpretation, if we assume that the initial values of the weights are close to 0. One can view weight decay as a kind of forgetting mechanism, which brings the weights closer to their initial values. This ensures that only the repeated updates have a significant effect on the absolute magnitude of the weights. A forgetting mechanism prevents a model from *memorizing* the training data, because only significant and repeated updates will be reflected in the weights.

5.4.1 Connections with Noise Injection

The addition of noise to the input has connections with penalty-based regularization. It can be shown that the addition of an equal amount of Gaussian noise to each input is equivalent to Tikhonov regularization of a single-layer neural network with an identity activation function (for linear regression).

One way of showing this result is by examining a single training case $(\overline{X}, y)$, which becomes $(\overline{X} + \sqrt{\lambda}\overline{\epsilon}, y)$ after noise with variance λ is added to each feature. Here, $\overline{\epsilon}$ is a random vector, in which each entry ϵ_i is independently drawn from the standard normal distribution with zero mean and unit variance. Then, the noisy prediction $\hat{y}$, which is based on $\overline{X} + \sqrt{\lambda}\overline{\epsilon}$, is as follows:

$$\hat{y} = \overline{W} \cdot (\overline{X} + \sqrt{\lambda}\epsilon) = \overline{W} \cdot \overline{X} + \sqrt{\lambda}\overline{W} \cdot \overline{\epsilon} \tag{5.7}$$

Now, let us examine the squared loss function $L = (y - \hat{y})^2$ contributed by a single training case. We will compute the *expected* value of the loss function. It is easy to show the following in expectation:

$$\begin{aligned} E[L] &= E[(y - \hat{y})^2] \\ &= E[(y - \overline{W} \cdot \overline{X} - \sqrt{\lambda}\overline{W} \cdot \overline{\epsilon})^2] \end{aligned}$$

One can then expand the expression on the right-hand side as follows:

$$\begin{aligned} E[L] &= (y - \overline{W} \cdot \overline{X})^2 - 2\sqrt{\lambda}(y - \overline{W} \cdot \overline{X}) \underbrace{E[\overline{W} \cdot \overline{\epsilon}]}_{0} + \lambda E[(\overline{W} \cdot \overline{\epsilon})^2] \\ &= (y - \overline{W} \cdot \overline{X})^2 + \lambda E[(\overline{W} \cdot \overline{\epsilon})^2] \end{aligned}$$

The second expression can be expanded using $\overline{\epsilon} = (\epsilon_1 \ldots \epsilon_d)$ and $\overline{W} = (w_1 \ldots w_d)$. Furthermore, one can set any term of the form $E[\epsilon_i \epsilon_j]$ to $E[\epsilon_i] \cdot E[\epsilon_j] = 0$ because of independence of the random variables ϵ_i and ϵ_j. Any term of the form $E[\epsilon_i^2]$ is set to 1, because each ϵ_i is drawn from a standard normal distribution. On expanding $E[(\overline{W} \cdot \overline{\epsilon})^2]$ and making the above substitutions, one finds the following:

$$E[L] = (y - \overline{W} \cdot \overline{X})^2 + \lambda(\sum_{i=1}^{d} w_i^2) \tag{5.8}$$

It is noteworthy that *this loss function is exactly the same as L_2-regularization* of a single instance.

Although the equivalence between weight decay and noise addition holds for the case of linear regression, the analysis is not quite exact in the case of neural networks with nonlinear activations. Nevertheless, penalty-based regularization continues to be intuitively similar to noise addition even in these cases. Because of these similarities one sometimes tries to perform regularization by direct noise addition. One such approach is referred to as *data perturbation*, in which noise is added to the training input, and the test data points are predicted with the added noise. This approach is described in section 5.5.5.

5.4.2 L_1-Regularization

The use of the squared norm penalty, which is also referred to as L_2-regularization, is the most common approach for regularization. However, it is possible to use other types of penalties on the parameters. A common approach is L_1-regularization in which the squared penalty is replaced with a penalty on the sum of the absolute magnitudes of the coefficients. Therefore, the new objective function is as follows:

$$J = L + \lambda \cdot \sum_{i=0}^{d} |w_i|_1 = \sum_{(x,y) \in \mathcal{D}} (y - \hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} |w_i|_1$$

The main problem with this objective function is that it contains the term $|w_i|$, which is not differentiable when w_i is exactly equal to 0. This requires some modifications to the gradient-descent method when w_i is 0. For the case when w_i is non-zero, one can use the straightforward update obtained by computing the partial derivative. By differentiating the above objective function, we can define the update equation at least for the case when w_i is different than 0:

$$w_i \Leftarrow w_i - \alpha\lambda s_i - \alpha \frac{\partial L}{\partial w_i}$$

The value of s_i, which is the partial derivative of $|w_i|$ (with respect to w_i), is as follows:

$$s_i = \begin{cases} -1 & w_i < 0 \\ +1 & w_i > 0 \end{cases}$$

However, we also need to set the partial derivative of $|w_i|$ for cases in which the value of w_i is exactly 0. One possibility is to use the *subgradient* method in which the value of w_i is set stochastically to a value in $\{-1, +1\}$. However, this is not necessary in practice. Computers are of finite-precision, and the computational errors will rarely cause w_i to be *exactly* 0. Therefore, the computational errors will often perform the task that would otherwise be achieved by stochastic sampling. Furthermore, for the rare cases in which the value w_i is exactly 0, one can omit the regularization and simply set s_i to 0. This type of approximation to the subgradient method works reasonably well in many settings.

One difference between the update equations for L_1-regularization and those in L_2-regularization is that L_2-regularization uses multiplicative decay as a forgetting mechanism, whereas L_1-regularization uses additive updates as a forgetting mechanism. In both cases, the regularization portions of the updates tend to move the coefficients closer to 0. However, they have different properties, which are discussed in the next section.

5.4.3 L_1- or L_2-Regularization?

A question arises as to whether L_1- or L_2-regularization is desirable. From an accuracy point of view, L_2-regularization usually outperforms L_1-regularization. This is the reason that L_2-regularization is almost always preferred over L_1-regularization is most implementations. The performance gap is small when the number of inputs and units is large.

However, L_1-regularization does have specific applications from an interpretability point of view. An interesting property of L_1-regularization is that it creates *sparse* solutions in which the vast majority of the values of w_i are 0s (after ignoring[1] computational errors). If the value of w_i is zero for a connection incident on the input layer, then that particular input has no effect on the final prediction. In other words, such an input can be *dropped*, and the L_1-regularizer acts as a feature selector. Therefore, one can use L_1-regularization to estimate which features are predictive to the application at hand.

What about the connections in the hidden layers whose weights are set to 0? These connections can be dropped, which results in a sparse neural network. Such sparse neural networks can be useful in cases where one repeatedly performs training on the same type of data set, but the nature and broader characteristics of the data set do not change significantly with time. Since the sparse neural network will contain only a small fraction of the connections in the original neural network, it can be retrained much more efficiently whenever more training data is received.

5.4.4 Penalizing Hidden Units: Learning Sparse Representations

The penalty-based methods, which have been discussed so far, penalize the *parameters* of the neural network. A different approach is to penalize the *activations* of the neural network, so that only a small subset of the neurons are activated for any given data instance. In other words, even though the neural network might be large and complex only a small part of it is used for predicting any given data instance.

[1]Computational errors can be ignored by requiring that $|w_i|$ should be at least 10^{-6} in order for w_i to be considered truly non-zero.

The simplest way to achieve sparsity is to impose an L_1-penalty on the hidden units. In other words, the loss on the ith hidden unit is $L_i^h = \lambda |h(i)|$, where λ is the regularization parameter. Therefore, the original loss function L is modified to the regularized loss function L^+ as follows:

$$L^+ = L + \sum_{i=1}^{M} L_i^h = L + \lambda \sum_{i=1}^{M} |h(i)| \tag{5.9}$$

Here, M is the total number of units in the network, and $h(i)$ is the value of the ith hidden unit.

As discussed in section 2.6.5 of Chapter 2, this change in loss function does not affect the overall dynamics and principles of backpropagation. The backpropagation algorithm needs to be modified so that the regularization penalty contributed by a hidden unit is incorporated into the backwards gradient flow at that node. Let $\Delta(i)$ be the derivative of the loss with respect to the pre-activation value in node i, and $A(i)$ be the set of nodes connected to node i using outgoing edges from node i. Furthermore, let Φ_i' be the numerical value of the local derivative of the activation function at node i, and w_{ij} be the weight of edges from node i to node j. We repeat Equation 2.35 from section 2.6.5, which shows how one can incorporate the effect of node penalties with an extra flow term to account for these penalties:

$$\Delta(i) \Leftarrow [\Phi_i' \sum_{j \in A(i)} w_{ij} \Delta(j)] + \underbrace{\Phi_i' \frac{\partial L_i^h}{\partial h(i)}}_{\text{Extra flow}} \tag{5.10}$$

The above update is specific to node i, since it computes the partial derivative of the loss with respect to the pre-activation variable at node i. The second term accounts for the node penalties. One can simplify this update to obtain the following:

$$\Delta(i) \Leftarrow [\Phi_i' \sum_{j \in A(i)} w_{ij} \Delta(j)] + \underbrace{\lambda \, \Phi_i' \, \text{sign}\{h(i)\}}_{\text{Extra flow}} \tag{5.11}$$

In other words, the only change that is required to a node-specific gradient update is to add the derivative of the regularization penalty at that node.

5.5 Ensemble Methods

Ensemble methods derive their inspiration from the bias-variance trade-off. One way of reducing the error of a classifier is to find a way to design the classifier to reduce the sum of the squared bias and the variance. Most ensemble methods in neural networks are focused on variance reduction. This is because neural networks are valued for their ability to build arbitrarily complex models in which the bias is relatively low. However, operating at the complex end of the bias-variance trade-off almost always leads to higher variance, which is manifested as overfitting. Therefore, this section will focus on such variance reduction methods.

5.5.1 Bagging and Subsampling

Imagine that you had an infinite resource of training data available to you, where you could generate as many training points as you wanted from a base distribution. A natural approach for reducing the variance in this case would be to repeatedly create different training data

sets and predict the same test instance using these data sets. The prediction across different data sets can then be averaged to yield the final prediction. If a sufficient number of training data sets is used, the variance of the prediction will be reduced to 0, although the bias will still remain depending on the choice of model.

The approach described above can be used only when an infinite resource of data is available. However, in practice, we only have a single finite instance of the data available to us. In such cases, one obviously cannot implement the above methodology. However, it turns out that an imperfect simulation of the above methodology still has better variance characteristics than a single execution of the model on the entire training data set. The basic idea is to generate new training data sets from the single instance of the base data by sampling. The sampling can be performed with or without replacement. The predictions on a particular test instance, which are obtained from the models built with different training sets, are then averaged to create the final prediction. One can average either the real-valued predictions (e.g., probability estimates of class labels) or the discrete predictions. In the case of real-valued predictions, better results are sometimes obtained by using the median of the values.

It is common to use the softmax to yield probabilistic predictions of discrete outputs. If probabilistic predictions are averaged, it is common to average the *logarithms* of these values. This is the equivalent of using the *geometric* means of the probabilities. For discrete predictions, arithmetically averaged voting is used. This distinction between the handling of discrete and probabilistic predictions is carried over to other types of ensemble methods that require averaging of the predictions. This is because the logarithms of the probabilities have a log-likelihood interpretation, and log-likelihoods are inherently additive.

The main difference between bagging and subsampling is in terms of whether or not replacement is used in the creation of the sampled training data sets. We summarize these methods as follows:

1. *Bagging:* In bagging, the training data is sampled with replacement. The sample size s may be different from the size of the training data size n, although the classical approach is to set s to n. In the latter case, the resampled data will contain duplicates, and about a fraction $(1 - 1/n)^n \approx 1/e$ of the original data set will not be included at all. Here, the notation e denotes the base of the natural logarithm. A model is constructed on the resampled training data set, and each test instance is predicted with the resampled data. The entire process of resampling and model building is repeated m times. For a given test instance, each of these m models is applied to the test data. The predictions from different models are then averaged to yield a single robust prediction. Although classical bagging uses $s = n$, the best results are often obtained by choosing values of s much less than n.

2. Subsampling is similar to bagging, except that the different models are constructed on the samples of the data created *without* replacement. The predictions from the different models are averaged. In this case, it is essential to choose $s < n$, because choosing $s = n$ yields the same training data set and identical results across different ensemble components.

When a sufficient training data are available, subsampling is often preferable to bagging. However, using bagging makes sense when the amount of available data is limited.

It is noteworthy that all the variance cannot be removed by using bagging or subsampling, because the different training samples will have overlaps in the included points. Therefore, the predictions of test instances from different samples will be positively correlated. The average of a set of random variables that are positively correlated will always

have a variance that increases with positive correlation. As a result, there will always be a residual variance in the predictions. This residual variance is a consequence of the fact that bagging and subsampling are imperfect simulations of drawing the training data from a base distribution. Nevertheless, the variance of this approach is still lower than that of constructing a single model on the entire training data set. The main challenge in directly using bagging for neural networks is that one must construct multiple training models, which is highly inefficient. However, the construction of different models can be fully parallelized, which makes the approach highly scalable.

5.5.2 Parametric Model Selection and Averaging

One challenge in the case of neural network construction is the selection of a large number of hyperparameters like the depth of the network and the number of neurons in each layer. Furthermore, the choice of the activation function also has an effect on performance, depending on the application at hand. The presence of a large number of parameters creates problems in model construction, because the performance might be sensitive to the particular configuration used. One possibility is to hold out a portion of the training data and try different combinations of parameters and model choices. The selection that provides the highest accuracy on the held-out portion of the training data is then used for prediction. This is, of course, the standard approach used for parameter tuning in all machine learning models, and is also referred to as *model selection*. In a sense, model selection is inherently an ensemble-centric approach, where the best out of bucket of models is selected. Therefore, the approach is also sometimes referred to as the *bucket-of-models* technique.

The main problem in deep learning settings is that the number of possible configurations is rather large. For example, one might need to select the number of layers, the number of units in each layer, and the activation function. The combination of these possibilities is rather large. Therefore, one is often forced to try only a limited number of possibilities to choose the configuration. An additional approach that can be used to reduce the variance, is to select the k best configurations and then average the predictions of these configurations. Such an approach leads to more robust predictions, especially if the configurations are very different from one another. Even though each individual configuration might be suboptimal, the overall prediction will still be quite robust. As in the case of bagging, the training of multiple configurations can be parallelized.

5.5.3 Randomized Connection Dropping

The random dropping of connections between different layers in a multilayer neural network often leads to diverse models in which different combinations of features are used to construct the hidden variables. The dropping of connections between layers does tend to create less powerful models because of the addition of constraints to the model-building process. However, since different random connections are dropped from different models, the predictions from different models are very diverse. The averaged prediction from these different models is often highly accurate. It is noteworthy that the weights of different models are not shared in this approach, which is different from another technique called *Dropout*.

Randomized connection dropping can be used for any type of predictive problem and not just classification. For example, the approach has been used for outlier detection with autoencoder ensembles [64]. As discussed in section 3.4.4 of Chapter 3, autoencoders can be used for outlier detection by estimating the reconstruction error of each data point. The work in [64] uses multiple autoencoders with randomized connections, and then aggregates

the outlier scores from these different components in order to create the score of a single data point. However, the use of the median is preferred to the mean in [64]. It has been shown in [64] that such an approach improves the overall accuracy of outlier detection. It is noteworthy that this approach might seem superficially similar to *Dropout* and *DropConnect*, although it is quite different. This is because methods like *Dropout* and *DropConnect* share weights between different ensemble components, whereas this approach does not share any weights between ensemble components.

5.5.4 Dropout

Dropout is a method that uses node sampling instead of edge sampling in order to create a neural network ensemble. If a node is dropped, then all incoming and outgoing connections from that node need to be dropped as well. The nodes are sampled only from the input and hidden layers of the network. Note that sampling the output node(s) would make it impossible to provide a prediction and compute the loss function. In some cases, the input nodes are sampled with a different probability from that of the hidden nodes. Therefore, if the full neural network contains M nodes, then the total number of possible sampled networks is 2^M.

A key point that is different from the connection sampling approach discussed in the previous section is that *weights of the different sampled networks are shared*. Therefore, *Dropout* combines node sampling with weight sharing. The training process then uses a single sampled example in order to update the weights of the sampled network using backpropagation. The training process proceeds using the following steps, which are repeated again and again in order to cycle through all of the training points in the network:

1. Sample a neural network from the base network. The input nodes are each sampled with probability p_i, and the hidden nodes are each sampled with probability p_h. Furthermore, all samples are independent of one another. When a node is removed from the network, all its incident edges are removed as well.

2. Sample a single training instance or a mini-batch of training instances.

3. Update the weights of the retained edges in the network using backpropagation on the sampled training instance or the mini-batch of training instances.

It is common to drop out nodes with probability between 20% and 50%. Large learning rates are often used with momentum, which are tempered with a max-norm constraint on the weights. In other words, the L_2-norm of the weights entering each node is constrained to be no larger than a small constant such as 3 or 4.

It is noteworthy that a different neural network is used for every small mini-batch of training examples. Therefore, the number of neural networks sampled is rather large, and depends on the size of the training data set. This is different from most other ensemble methods like bagging in which the number of ensemble components is rarely larger than 25. In the *Dropout* method, thousands of neural networks are sampled with shared weights, and a tiny training data set is used to update the weights in each case. Even though a large number of neural networks is sampled, the *fraction* of neural networks sampled out of the base number of possibilities is still minuscule. Another assumption that is used in this class of neural networks is that the output is in the form of a probability. This assumption has a bearing on the way in which the predictions of the different neural networks are combined.

How can one use the ensemble of neural networks to create a prediction for an unseen test instance? One possibility is to predict the test instance using all the neural networks

that were sampled, and then use the geometric mean of the probabilities that are predicted by the different networks. The geometric mean is used rather than the arithmetic mean, because the assumption is that the output of the network is a probability and the geometric mean is equivalent to averaging log-likelihoods. For example, if the neural network has k probabilistic outputs corresponding to the k classes, and the jth ensemble yields an output of $p_i^{(j)}$ for the ith class, then the ensemble estimate for the ith class is computed as follows:

$$p_i^{Ens} = \left[\prod_{j=1}^{m} p_i^{(j)}\right]^{1/m} \tag{5.12}$$

Here, m is the total number of ensemble components, which can be rather large in the case of the *Dropout* method. One problem with this estimation is that the use of geometric means results in a situation where the probabilities over the different classes do not sum to 1. Therefore, the values of the probabilities are re-normalized so that they sum to 1:

$$p_i^{Ens} \Leftarrow \frac{p_i^{Ens}}{\sum_{i=1}^{k} p_i^{Ens}} \tag{5.13}$$

The main problem with this approach is that the number of ensemble components is too large, which makes the approach inefficient.

A key insight of the *Dropout* method is that it is not necessary to evaluate the prediction on all ensemble components. Rather, one can perform forward propagation on only the base network (with no dropping) after re-scaling the weights. The basic idea is to multiply the weights going out of each unit with the probability of sampling that unit. By using this approach, the expected output of that unit from a sampled network is captured. This rule is referred to as the *weight scaling inference rule*. Using this rule also ensures that the input going into a unit is also the same as the expected input that would occur in a sampled network.

The weight scaling inference rule is exact for many types of networks with linear activations, although the rule is not exactly true for networks with nonlinearities. In practice, the rule tends to work well across a broad variety of networks. Since most practical neural networks have nonlinear activations, the weight scaling inference rule of *Dropout* should be viewed as a heuristic rather than a theoretically justified result.

The main effect of *Dropout* is to incorporate regularization into the learning procedure. By dropping both input units and hidden units, *Dropout* effectively incorporates noise into both the input data and the hidden representations. The nature of this noise can be viewed as a kind of masking noise in which some inputs and hidden units are set to 0. Noise addition is a form of regularization. It has been shown in the original paper [483] on *Dropout* that this approach works better than other regularizers such as weight decay. *Dropout* prevents a phenomenon referred to as *feature co-adaptation* from occurring between hidden units. Since the effect of *Dropout* is a masking noise that removes some of the hidden units, this approach forces a certain level of redundancy between the features learned at the different hidden units. This type of redundancy leads to increased robustness.

Dropout is efficient because each of the sampled subnetworks is trained with a small set of sampled instances. Therefore, only the work of sampling the hidden units needs to be done additionally. However, since *Dropout* is a regularization method, it reduces the expressive power of the network. Therefore, one needs to use larger models and more units in order to gain the full advantages of *Dropout*. This results in a hidden computational

overhead. Furthermore, if the original training data set is already large enough to reduce the likelihood of overfitting, the additional computational advantages of *Dropout* may be small but still perceptible. For example, many of the convolutional neural networks trained on large data repositories like *ImageNet* [263] report consistently improved results of about 2% with *Dropout*. A variation of *Dropout* is *DropConnect*, which applies a similar approach to the weights rather than to the neural network nodes [531].

A Note on Feature Co-Adaptation

In order to understand why *Dropout* works, it is useful to understand the notion of feature co-adaptation. Ideally, it is useful for the hidden layers of the neural network to create features that reflect important classification characteristics of the input without having complex dependencies on other features, unless these other features are truly useful. To understand this point, consider a situation in which all edges incident on 50% of the nodes in each layer are fixed at their initial random values, and are not *updated* during backpropagation (even though all gradients are *computed* in the normal fashion). Interestingly, even in this case, it will often be possible for the neural network to provide reasonably good results by adapting the other weights and features to the effect of these randomly fixed subsets of weights (and corresponding activations). Of course, this is not a desirable situation because the goal of features working together is to combine the powers held by each essential feature rather than merely having some features adjust to the detrimental effects of others. Even in the normal training of a neural network (where all weights are updated), this type of co-adaptation can occur. For example, if the updates in some parts of the neural network are not fast enough, some of the features will not be useful and other features will adapt to these less-than-useful features. This situation is very likely in neural network training, because different parts of the neural network do tend to learn at different rates. An even more troubling scenario arises when the co-adapted features work well in predicting training points by picking up on complex dependencies in the training points, which do not generalize well to out-of-sample test points. *Dropout* prevents this type of co-adaptation by forcing the neural network to make predictions using only a subset of the inputs and activations. This forces the network to be able to make predictions with a certain level of redundancy while also encouraging smaller subsets of learned features to have predictive power. In other words, co-adaptation occurs only when it is truly essential for modeling instead of learning random nuances of the training data. This is, of course, a form of regularization. Furthermore, by learning redundant features, *Dropout* averages over the predictions of redundant features, which is similar to what is done in bagging.

5.5.5 Data Perturbation Ensembles

Many variance reduction methods like penalty-based regularization and *Dropout* are equivalent to noise injection. Data perturbation *explicitly* injects noise to the training data. In the simplest case, a small amount of noise can be added to the input data, and the weights can be learned on the perturbed data. This process can be repeated with multiple such additions to the training data, and it does not need to be added to the test data. When explicitly adding noise, it is important to average the prediction of the same test instance over multiple runs in order to ensure that the solution properly represents the *expected* value of the loss (without added variance caused by the noise). This type of approach is a generic ensemble method, which is not specific to neural networks. As discussed in section 5.10, this approach is used commonly in the unsupervised setting with *de-noising autoencoders*. It is

also possible to add noise to the hidden layer. However, in this case, the noise has to be carefully calibrated [396]. It is noteworthy that the *Dropout* method indirectly adds noise to the hidden layer by dropping nodes randomly. A dropped node is similar to masking noise in which the activation of that node is set to 0.

One can also perform other types of data set augmentation. For example, an image instance can be rotated or translated in order to add to the data set. Carefully designed data augmentation schemes can often greatly improve the accuracy of a learner by increasing its generalization power. However, strictly speaking such schemes are not perturbation schemes because the augmented examples are created with a calibrated procedure and an understanding of the domain at hand. Such methods are used commonly in convolutional neural networks (cf. section 9.3.4 of Chapter 9).

5.6 Early Stopping

Neural networks are trained using variations of gradient-descent methods. In most optimization models, gradient-descent methods are executed to convergence. However, executing gradient descent to convergence optimizes the loss on the training data, but not necessarily on the out-of-sample test data. This is because the final few steps often overfit to the specific nuances of the training data, which might not generalize well to the test data.

A natural solution to this dilemma is to use *early stopping*. In this method, a portion of the training data is held out as a validation set. The backpropagation-based training is only applied to the portion of the training data that does not include the validation set. At the same time, the error of the model on the validation set is continuously monitored. At some point, this error begins to rise on the validation set, even though it continues to reduce on the training set. This is the point at which further training causes overfitting. Therefore, this point can be chosen for termination. It is important to keep track of the best solution achieved so far in the learning process (as computed on the validation data). This is because one does not perform early stopping after tiny increases in the out-of-sample error (which might be caused by noisy variations), but it is advisable to continue to train to check if the error continues to rise. In other words, the termination point is chosen in hindsight after the error on the validation set continues to rise, and all hope is lost of improving the error performance on the validation set.

One advantage of early stopping is that it can be easily added to neural network training without significantly changing the training procedure. Furthermore, methods like weight decay require us to try different values of the regularization parameter, λ, which can be expensive. Because of the ease in combining it with existing algorithms, early stopping can be used in combination with other regularizers in a relatively straightforward way. Therefore, early stopping is almost always used, because one does not lose much by adding it to the learning procedure.

One can view early stopping as a kind of constraint on the optimization process. By restricting the number of steps in the gradient descent, one is effectively restricting the distance of the final solution from the initialization point. Adding constraints to the model of a machine learning problem is often a form of regularization.

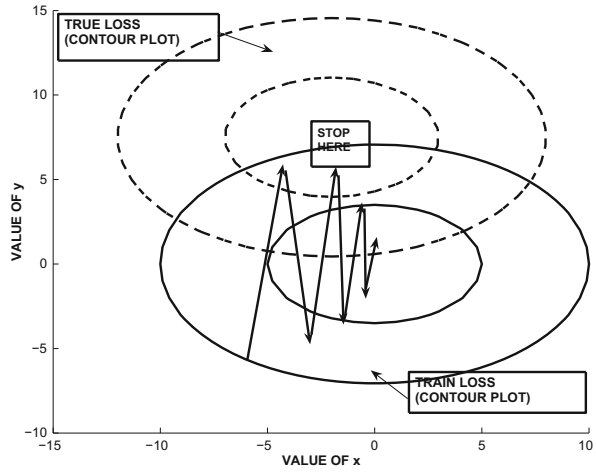


Figure 5.6: Shift in loss function caused by variance effects and the effect of early stopping. Because of the differences in the true loss function and that on the training data, the error will begin to rise if gradient descent is continued beyond a certain point. Here, we have shown a similar shape of the true and training loss functions for simplicity, although this might not be the case in practice.

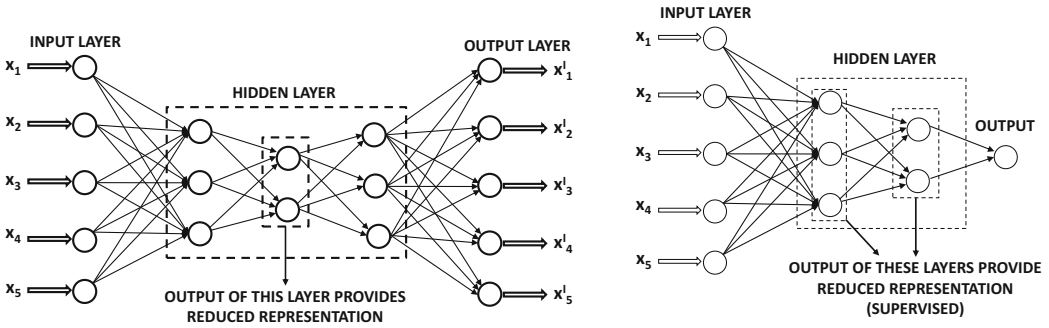
5.6.1 Understanding Early Stopping from the Variance Perspective

One way of understanding the bias-variance trade-off is that the true loss function of an optimization problem can only be constructed if we have infinite data. If we have a finite amount of data, the loss function constructed from the training data does not reflect the true loss function. Illustrative examples of the contours of the true loss function and its shifted counterpart on the training data are illustrated in Figure 5.6. This shifting is an indirect manifestation of the variance in prediction created by a particular training data set. Different training data sets will shift the loss function in different and unpredictable ways.

Unfortunately, the learning procedure can perform the gradient-descent only on the loss function defined on the training data set, because the true loss function is unknown. However, if the training data is representative of the true loss function, the optimum solutions in the two cases will be reasonably close as shown in Figure 5.6. As discussed in Chapter 4, most gradient-descent procedures take a circuitous and oscillatory route to the optimal solution. During the final stage of convergence to the optimal solution (on the training data), the gradient descent will often encounter better solutions with respect to the true loss function before it converges to the best solution with respect to the training data. These solutions will be detected by the improved accuracy on the validation set, and therefore provide good termination points. An example of a good early stopping point is shown in Figure 5.6.

5.7 Unsupervised Pretraining

Deep networks are inherently hard to train because of a number of different characteristics discussed in the previous chapter. One issue is the exploding and vanishing gradient problem,



(a) Multilayer autoencoder (b) Multilayer classifier

Figure 5.7: Both the multilayer classifier and the multilayer autoencoder use a similar pretraining procedure.

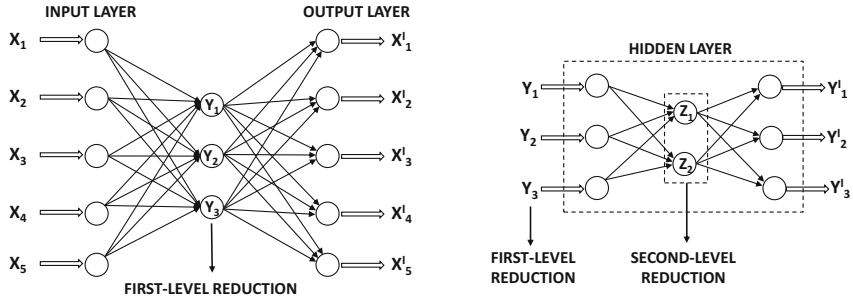
because of which the different layers of the neural network do not get trained at the same rate. The multiple layers of the neural network cause distortions in the gradient, which make them hard to train.

A ground-breaking break-through in this context was the use of unsupervised pretraining in order to provide robust initializations [206]. This initialization is achieved by training the network greedily in layer-wise fashion. The approach was originally proposed in the context of deep belief networks, but it was later extended to other types of models such as autoencoders [402, 526]. In this chapter, we will study the autoencoder approach because of its simplicity. First, we will start with the dimensionality reduction application, because the application is unsupervised and it is easy to show how to use unsupervised pretraining in this case. However, unsupervised pretraining can also be used for supervised applications like classification with minor modifications.

In pretraining, a greedy approach is used to train the network one layer at a time by learning the weights of the outer hidden layers first and then learning the weights of the inner hidden layers. The resulting weights are used as starting points for a final phase of traditional neural network backpropagation in order to fine-tune them.

Consider the autoencoder and classifier architectures shown in Figure 5.7. Since these architectures have multiple layers, randomized initialization can sometimes cause challenges. However, it is possible to create a good initialization by setting the initial weights layer by layer in a greedy fashion. First, we describe the process in the context of the autoencoder shown in Figure 5.7(a), although an almost identical procedure is relevant to the classifier of Figure 5.7(b). We have intentionally chosen neural architectures in the two cases so that the hidden layers have similar numbers of nodes.

The pretraining process is shown in Figure 5.8. The basic idea is to assume that the two (symmetric) outer hidden layers contain a first-level reduced representation of larger dimensionality, and the inner hidden layer contains a second-level reduced representation of smaller dimensionality. Therefore, the first step is to learn the first-level reduced representation and the corresponding weights associated with the outer hidden layers using the simplified network of Figure 5.8(a). In this network, the middle hidden layer is missing and the two outer hidden layers are collapsed into a single hidden layer. The assumption is that the two outer hidden layers are related to one another in a symmetric way like a smaller autoencoder. In the second step, the reduced representation in the first step is used to learn the second-level reduced representation (and weights) of the inner hidden layers. Therefore,



(a) Pretraining first-level reduction and outer weights

(b) Pretraining second-level reduction and inner weights

Figure 5.8: Pretraining a neural network

the inner portion of the neural network is treated as a smaller autoencoder in its own right. Since each of these pretrained subnetworks is much smaller, the weights can be learned more easily. This initial set of weights is then used to train the entire neural network with backpropagation. Note that this process can be performed in layerwise fashion for a deep neural network containing any number of hidden layers.

So far, we have only discussed how we can use unsupervised pretraining for unsupervised applications. A natural question arises as to how one can use pretraining for supervised applications. Consider a multilayer classification architecture with a single output layer and k hidden layers. During the pretraining stage, the output layer is removed, and the representation of the final hidden layer is learned in an unsupervised way. This is achieved by creating an autoencoder with $2 \cdot k - 1$ hidden layers, where the middle layer is the final hidden layer of the supervised setting. For example, the relevant autoencoder for Figure 5.7(b) is shown in Figure 5.7(a). Therefore, an additional $(k-1)$ hidden layers are added, each of which has a symmetric counterpart in the original network. This network is trained in exactly the same layer-wise fashion as discussed above for the autoencoder architecture. The weights of only the encoder portion of this autoencoder are used for initialization of the weights entering into all hidden layers. The weights between the final hidden layer and the output layer can also be initialized by treating the final hidden layer and output nodes as a single-layer network. This single-layer network is fed with the reduced representations of the final hidden layer (based on the autoencoder learned in pretraining). After the weights of all the layers have been learned, the output nodes are re-attached to the final hidden layer. The backpropagation algorithm is applied to this initialized network in order to fine-tune the weights from the pretrained stage. Note that this approach learns all the initial hidden representations in an unsupervised way, and only the weights entering into the output layer are initialized using the labels. Therefore, the pretraining can still be considered to be largely unsupervised.

During the early years, pretraining was often seen as a more stable way to train a deep network in which the different layers have a better chance of being initialized in an equally effective way. Although this issue does play a role in explaining the improvements of pretraining, the problem is often manifested as overfitting. As discussed in Chapter 4, the (finally converged) weights in the early layers may not change much from their random initializations, when the network exhibits the vanishing gradient problem. Even when the connection weights in the first few layers are random (as a result of poor training), it is possible for the later layers to adapt their weights sufficiently so as to give zero error on the

training data. In this case, the random connections in the early layers provide near-random transformations to the later layers, but the later layers are still able to overfit to these features in order to provide very low training error. In other words, the features in later layers *adapt* to those in early layers as a result of training inefficiencies. Any kind of feature co-adaptation caused by training inefficiencies almost always leads to overfitting. Therefore, when the approach is applied to unseen test data, the overfitting becomes apparent because the various layers are not specifically adapted to these unseen test instances. In this sense, pretraining is an unusual form of regularization.

Incidentally, unsupervised pretraining helps even in cases where the amount of training data is very large. It is likely that this behavior is caused by the fact that pretraining helps in issues beyond model generalization. One evidence of this fact is that in larger data sets, even the error on the training data seems to be high, when methods like pretraining are not used. In these cases, the weights of the early layers often do not change much from their initializations, and one is using only a small number of later layers on a random transformation of the data (defined by the random initialization of the early layers). As a result, the trained portion of the network is rather shallow, with some additional loss caused by the random transformation. In such cases, pretraining also helps a model realize the full benefits of depth, thereby facilitating the improvement of prediction accuracy on larger data sets.

Another way of understanding pretraining is that it provides insights into the repeated patterns in the data, which are the features learned from the training data points. For example, an autoencoder might learn that many digits have loops in them, and certain digits have strokes that are curved in a particular way. The decoder reconstructs the digits by putting together these frequent shapes. However, these shapes also have discriminative power with respect to recognizing digits. Expressing the data in terms of a few features then helps in recognizing how these features are related to the class labels. This principle is summarized by Geoff Hinton [202] in the context of image classification as follows: "*To recognize shapes, first learn to generate images.*" This type of regularization preconditions the training process in a semantically relevant region of the parameter space, where several important features have already been learned, and further training can fine-tune and combine them for prediction.

5.7.1 Variations of Unsupervised Pretraining

There are many different ways in which one can introduce variations to the procedure of unsupervised pretraining. For example, multiple layers can be trained at one time instead of performing pretraining only one layer at a time. A particular case in point is *VGG* (cf. section 9.4.3 of Chapter 9) in which as many as eleven layers of an even deeper architecture were trained together. Indeed, there are some advantages in grouping as many layers as possible within the pretraining because a (successful) training procedure with larger pieces of the neural network leads to more powerful initializations. On the other hand, grouping too many layers together within each pretraining component can lead to problems (such as the vanishing and exploding gradient problems) within each component.

A second point is that the pretraining procedure of Figure 5.8 assumes that the autoencoder works in a completely symmetric way in which the reduction in the kth layer of the encoder is approximately similar to the reduction in its mirror layer in the decoder. This might be a restrictive assumption in practice, if different types of activation functions are used in different layers. For example, a sigmoid activation function in a particular layer of the encoder will create only nonnegative values, whereas a tanh activation in the matching

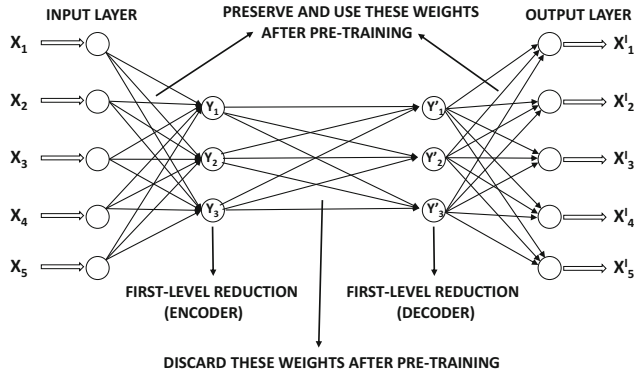


Figure 5.9: This architecture allows the first-level representations in the encoder and decoder to be significantly different. It is helpful to compare this architecture with that in Figure 5.8(a).

layer of the decoder might create both positive and negative values. Another approach is to use a relaxed pretraining architecture in which we learn separate reductions for the kth level reduction in the encoder and its mirror image in the decoder. This allows the corresponding reductions in the encoder and the decoder to be different. An additional layer of weights must be added between the two layers to allow for the differences between the two reductions. This additional layer of weights is discarded after the reduction, and only the encoder-decoder weights are preserved. The only location at which an additional set of weights is not used for pretraining is in the innermost reduction, which proceeds in a similar manner to that discussed in the earlier section (cf. Figure 5.8(b)). An example of such an architecture for the first-level reduction of Figure 5.8(a) is shown in Figure 5.9. Note that the first-level representations for the encoder and decoder layers can be quite different in this case, which provides some flexibility during the pretraining process. When the approach is used for classification, only the weights in the encoder can be used, and the final reduced code can be capped with a classification layer for learning.

5.7.2 What About Supervised Pretraining?

So far, we have only discussed *unsupervised* pretraining, whether the base application is supervised or unsupervised. Even in the case where the base application is supervised, the initialization was done using an unsupervised autoencoder architecture. Although it is possible to perform supervised pretraining as well, an interesting and surprising result is that supervised pretraining does not seem to give as good results as unsupervised pretraining in at least some settings [31, 116]. This does not mean that supervised pretraining is *never* helpful. Indeed, there are cases of networks in which it is hard to train the network itself because of its depth. For example, networks with hundreds of layers are extremely hard to train because of issues associated with convergence and other problems. In such cases, even the error *on the training data* is high, which means that one is unable to make the training algorithm work. *This is a different problem from that of model generalization.* Aside from supervised pretraining, many techniques such as the construction of *highway networks* [168, 486], *gating networks* [214], and *residual networks* [194], can address many of these problems. However, these solutions do not specifically address overfitting, whereas unsupervised pretraining seems to hedge its bets in addressing both issues in at least some types of networks.

In supervised pretraining [31], the autoencoder architecture is not used for learning the weights of connections incident on the hidden layer. In the first iteration, the constructed network contains only the first hidden layer, which is connected to all nodes in the output layer. This step learns the weights of the connections from the input to hidden layer, although the weights of the output layer are discarded. Subsequently, the outputs of the first hidden layer are used as the new representations of the training points. Then, we create another neural network containing the first and second hidden layers and the output layer. The first hidden layer is now treated as an input layer with its inputs as the transformed representations of the training points learned in the previous iteration. These are then used to learn the next layer of weights and their hidden representations. This approach is repeated all the way to the final layer. Although this approach does provide improvements over an approach that does not use pretraining, it does not seem to work as well as unsupervised pretraining in at least some settings. The main difference in performance is on the *generalization error* on unseen test data, whereas the errors on the training data are often similar [31]. This is a near-certain sign of differential levels of overfitting of different methods.

Why does supervised pretraining not help as much as unsupervised pretraining in many settings? A key problem of supervised pretraining is that it is a bit too greedy and the early layers are initialized to representations that are very directly related to the outputs. As a result, the full advantages of depth are not exploited. This is a different type of overfitting. An important explanation for the success of unsupervised pretraining is that the learned representations are often related to the class labels in a gentle way; as a result, further learning is able to isolate and fine-tune the important characteristics of these representations. Therefore, one can view pretraining as an unusual form of *semi-supervised learning* as well, which forces the initial representations of the hidden layers to lie on the low-dimensional manifolds of data instances. The secret to the success of pretraining is that more features on these manifolds are predictive of classification accuracy than the features corresponding to random regions of the data space. After all, class distributions vary smoothly over the underlying data manifolds. The locations of data points on these manifolds are therefore good features in predicting class distributions. Therefore, the final phase of learning only has to fine-tune and enhance these features.

5.8 Continuation and Curriculum Learning

The discussions in the previous and current chapter show that the learning of neural network parameters is inherently a complex optimization problem, in which the loss function has a complex topological shape. Furthermore, the loss function on the training data is not exactly the same as the true loss function, which leads to spurious minima. These minima are spurious because they might be near optimal minima on the training data, but they might not be minima at all on unseen test instances. In many cases, optimizing a complex loss function tends to lead to such solutions with little generalization power.

The experience with pretraining shows that simplifying the optimization problem (or providing simple greedy solutions without too much optimization) can often precondition the solution towards the basins of better optima on the test data. In other words, instead of trying to solve a complex problem in one shot, one should first try to solve simplifications, and gradually work one's way towards complex solutions. Two such notions are those of *continuation* and *curriculum* learning:

1. *Continuation learning:* In continuation learning, one starts with a simplified version of the optimization problem and solves it. Starting with this solution, one continues to a more complex refinement of the optimization problem and updates the solution. This process is repeated until the complex optimization problem is solved. Thus, continuation learning leverages a model-centric view of working from simpler to complex problems. For example, if one has a loss function with many local optima, one can smooth it to a loss function with a single global optimum and find the optimal solution. Then, one can gradually work with better and better approximations (with increased complexity) until the exact loss function is used.

2. *Curriculum learning:* In curriculum learning, one starts by training the model on simpler data instances, and then gradually adds more difficult instances to the training data. Therefore, curriculum learning leverages a data-centric view of working from the simple to the complex, whereas continuation methods leverage a model-centric view.

A different view of curriculum and continuation learning may be obtained by examining how humans naturally learn tasks. Humans often learn simple concepts first and then move to the complex. The training of a child is often created using such a *curriculum* in order to accelerate learning. This principle also seems to work well in machine learning. In the following, we will examine both continuation and curriculum learning.

Continuation Learning

In continuation learning, one designs a series of loss functions $L_1 \ldots L_r$, in which the difficulty in optimizing this sequence of loss functions grows from the easy to the difficult. In other words, each L_{i+1} is more difficult to optimize than L_i. All the optimization problems are defined on the same set of parameters, because they are defined on the same neural network. The smoothing of a loss function is a form of regularization. One can view each L_i as a smoothed version of L_{i+1}. Solving each L_i brings the solution closer to the basin of optimal solutions from the point of view of generalization error.

Continuation loss functions are often constructed by using *blurring*. The idea is to compute the loss function at sampled points in the vicinity of a given point, and then average these values in order to create the new loss function. For example, one could use a normal distribution with standard deviation σ_i for computing the ith loss function L_i. One can view this approach as a type of noise addition to the loss function, which is also a form of regularization. The amount of blurring depends on the size of the locality used for blurring, which is defined by σ_i. If the value of σ_i is set to be too large, then the cost will be very similar at all points, and the loss function will not retain sufficient details about the objective. However, it will often be very simple to optimize. On the other hand, setting σ_i to 0 will retain all the details in the loss function. Therefore, the natural solution is to start with large values of σ_i and then reduce the value over successive loss functions. One can view this approach as that of using an increased amount of noise for regularization in the early iterations, and then reducing the level of regularization as the algorithm nears an attractive solution. Such tricks of adding a varying amount of calibrated noise to enable the avoidance of local optima is a recurring theme in many optimization techniques such as *simulated annealing* [254]. The main problem with continuation methods is that they are expensive due to the need to optimize a series of loss functions.

Curriculum Learning

Curriculum learning methods take a *data-centric* view of the goals that are achieved by the *model-centric* continuation learning methods. The main hypothesis is that different training data sets present different levels of difficulty to a learner. In curriculum methods, easy examples are first presented to the learner. One possible way of defining a difficult example is as one that falls on the wrong side of a decision boundary with a perceptron or an SVM. There are other possibilities, such as the use of a Bayes classifier. The basic idea is that the difficult examples are often noisy or they represent exceptional patterns that confuse the learner. Therefore, it is inadvisable to start training with such examples.

In other words, the initial iterations of stochastic gradient descent use only the easy examples to "pretrain" the learner towards a reasonable parameter setting. Subsequently, difficult examples are included with the easy examples in later iterations. It is important to include both easy and difficult examples in the later phases, or else the learner will overfit to only the difficult examples. In many cases, the difficult examples might be exceptional patterns in particular regions of the space, or they might even be noise. If only the difficult examples are presented to the learner in later phases, the overall accuracy will not be good. The best results are often obtained by using a random mixture of simple and difficult examples in later phases. The proportion of difficult examples are increased over the course of the curriculum until the input represents the true data distribution. This type of *stochastic curriculum* has been shown to be an effective approach.

5.9 Parameter Sharing

A natural form of regularization that reduces the parameter footprint of the model is the sharing of parameters across different connections. Often, this type of parameter sharing is enabled by domain-specific insights. The main insight required to share parameters is that the function computed at two nodes should be related in some way. This type of insight can be obtained when one has a good idea of how a particular computational node relates to the input data. Examples of such parameter-sharing methods are as follows:

1. *Sharing weights in autoencoders:* The symmetric weights in the encoder and decoder portion of the autoencoder are often shared. Although an autoencoder will work whether or not the weights are shared, doing so improves the generalization properties of the algorithm to out-of-sample data.

2. *Recurrent neural networks:* These networks are often used for modeling sequential data, such as time-series, biological sequences, and text. In recurrent neural networks, a time-layered representation of the network is created in which the neural network is replicated across layers associated with time stamps. Since each time stamp is assumed to use the same model, the parameters are shared between different layers. Recurrent neural networks are discussed in detail in Chapter 8.

3. *Convolutional neural networks:* Convolutional neural networks are used for image recognition and prediction. Correspondingly, the inputs of the network are arranged into a rectangular grid pattern, along with all the layers of the network. Furthermore, the weights across contiguous patches of the network are typically shared. The basic idea is that a rectangular patch of the image corresponds to a portion of the visual field, and it should be interpreted in the same way no matter where it is located. In other words, a carrot means the same thing whether it is at the left or the right of

the image. In essence, these methods use semantic insights about the data to reduce the parameter footprint, share weights, and sparsify the connections. Convolutional neural networks are discussed in Chapter 9.

In many of these cases, it is evident that parameter sharing is enabled by the use of domain-specific insights about the training data as well as a good understanding of how the computed function at a node relates to the training data. The modifications to the backpropagation algorithm required for enabling weight sharing are discussed in section 2.6.6 of Chapter 2.

An additional type of weight sharing is *soft weight sharing* [371]. In soft weight sharing, the parameters are not completely tied, but a penalty is associated with them being different. For example, if one expects the weights w_i and w_j to be similar, the penalty $\lambda(w_i - w_j)^2/2$ might be added to the loss function. In such a case, the quantity $\alpha\lambda(w_j - w_i)$ might be added to the update of w_i, and the quantity $\alpha\lambda(w_i - w_j)$ might be added to the update of w_j. Here, α is the learning rate. These types of changes to the updates tend to pull the weights towards each other.

5.10 Regularization in Unsupervised Applications

Although overfitting does occur in unsupervised applications, it is often less of a problem. In classification, one is trying to learn a single bit of information associated with each example, and therefore using more parameters than the number of examples can cause overfitting. This is not quite the case in unsupervised applications in which a single training example may contain many more bits of information corresponding to the different dimensions. In general, the number of bits of information will depend on the intrinsic dimensionality of the data set. Therefore, one tends to hear fewer complaints about overfitting in unsupervised applications. Nevertheless, there are many unsupervised settings in which it is beneficial to use regularization. These settings will be discussed in this section.

5.10.1 When the Hidden Layer is Broader than the Input Layer

The discussion of autoencoders in Chapter 3 assumes that the hidden layer has fewer units than the input layer. It makes sense for the hidden layer to have fewer units than the input layer when one is looking for a compressed representation of the data. A constricted hidden layer forces dimensionality reduction, and the loss function is designed to avoid information loss. Such representations are referred to as *undercomplete representations*, and they correspond to the traditional use-case of autoencoders.

What about the case when the number of hidden units is greater than the input dimensionality? This situation corresponds to the case of *over-complete representations*. Increasing the number of hidden units beyond the number of input units makes it possible for the hidden layer to simply learn the identity function (with zero loss). Simply copying the input across the layers does not seem to be particularly useful. However, this does not occur in practice (while learning weights), especially if certain types of regularization and *sparsity constraints* are imposed on the hidden layer. Even if no sparsity constraints are imposed, and stochastic gradient descent is used for learning, the probabilistic regularization caused by stochastic gradient descent is sufficient to ensure that the hidden representation will always scramble the input before reconstructing it at the output. This is because stochastic

gradient descent is a type of noise addition to the learning process, and therefore it will not be possible to learn weights that simply copy input to output as identity functions across layers. Furthermore, because of some peculiarities of the training process, a neural network almost never uses its full modeling ability, which leads to dependencies among the weights [96]. Rather, an over-complete representation may be created, although it may not have the property of sparsity (which needs to be explicitly encouraged). The next section will discuss ways of encouraging sparsity.

5.10.1.1 Sparse Feature Learning

When explicit sparsity constraints are imposed, the resulting autoencoder is referred to as a *sparse autoencoder*. A sparse representation of a d-dimensional point is a k-dimensional point in which $k \gg d$ and most of the values in the sparse representation are 0s. Sparse feature learning has tremendous applicability to many settings like image data, where the learned features are often intuitively more interpretable from an application-specific perspective. Sparse representations also enable the effective use of particular types of efficient algorithms that are highly dependent on sparsity. There are many ways in which constraints might be enforced on the hidden layer to create sparsity. Some examples are as follows:

1. One can impose an L_1-penalty on the activations in the hidden layer to force sparse activations. The notion of L_1-penalties for creating sparse solutions (in terms of either weights or hidden units) is discussed in section 5.4.2 and 5.4.4 of Chapter 5. In such a case, backpropagation must also propagate the gradient of this penalty in the backwards direction. Surprisingly, this natural alternative is rarely used.

2. One can allow only the top-r activations in the hidden layer to be nonzero for $r \leq k$. In such a case, backpropagation only backpropagates through the activated units. This approach is referred to as the r-sparse autoencoder [321].

3. Another approach is the *winner-take-all* autoencoder [322], in which only a fraction f of the activations of *each* hidden unit are allowed over the *whole training data*. In this case, the top activations are computed across training examples, whereas in the previous case the top activations are computed across a hidden layer for a single training example. Therefore node-specific thresholds need to be estimated using the statistics of a minibatch. The backpropagation algorithm needs to propagate the gradient only through the activated units.

Note that the implementations of the competitive mechanisms are almost like ReLU activations with adaptive thresholds. Refer to the bibliographic notes for pointers and more details of these algorithms.

5.10.2 Noise Injection: De-noising Autoencoders

As discussed in section 5.4.1, noise injection is a form of penalty-based regularization of the weights. The use of Gaussian noise in the input is roughly equal to L_2-regularization in single-layer networks with linear activation. The de-noising autoencoder is based on noise injection rather than penalization of the weights or hidden units. However, the goal of the de-noising autoencoder is to reconstruct good examples from corrupted training data. Therefore, the type of noise should be calibrated to the nature of the input. Several different types of noise can be added:

1. *Gaussian noise:* This type of noise is appropriate for real-valued inputs. The added noise has zero mean and variance $\lambda > 0$ for each input. Here, λ is the regularization parameter.

2. *Masking noise:* The basic idea is to set a fraction f of the inputs to zeros in order to corrupt the inputs. This type of approach is particularly useful when working with binary inputs.

3. *Salt-and-pepper noise:* In this case, a fraction f of the inputs are set to either their minimum or maximum possible values according to a fair coin flip. The approach is typically used for binary inputs, for which the minimum and maximum values are 0 and 1, respectively.

De-noising autoencoders are useful when dealing with data that is corrupted. Therefore, the main application of such autoencoders is to reconstruct corrupted data. The inputs to the autoencoder are corrupted training records, and the outputs are the uncorrupted data records. As a result, the autoencoder learns to recognize the fact that the input is corrupted, and the true representation of the input needs to be reconstructed. Therefore, even if there is corruption in the test data (as a result of application-specific reasons), the approach is able to reconstruct clean versions of the test data. Note that the noise in the training data is explicitly added, whereas that in the test data is already present as a result of various application-specific reasons. For example, as shown in the top portion of Figure 5.10, one can use the approach to removing blurring or other noise from images. The nature of the noise added to the input training data should be based on insights about the type of corruption present in the test data. Therefore, one does require uncorrupted examples of the training data for best performance. In most domains, this is not very difficult to achieve. For example, if the goal is to remove noise from images, the training data might contain high-quality images as the output and artificially blurred images as the input. It is common for the de-noising autoencoder to be overcomplete, when it is used for reconstruction from corrupted data. However, this choice also depends on the nature of the input and the amount of noise added. Aside from its use for reconstructing inputs, the addition of noise is also an excellent regularizer that tends to make the approach work better for out-of-sample inputs even when the autoencoder is undercomplete.

The way in which the de-noising autoencoder works is that it uses the noise in the input data to learn the true manifold on which the data is embedded. Each corrupted point is projected to its "closest" matching point on the true manifold of the data distribution. The closest matching point is the expected position on the manifold from which the model predicts that the noisy point has originated. This projection is shown in the bottom portion of Figure 5.10. The true manifold is a more concise representation of the data as compared to the noisy data, and this conciseness is a result of the regularization inherent in the addition of noise to the input. All forms of regularization tend to increase the conciseness of the underlying model.

5.10.3 Gradient-Based Penalization: Contractive Autoencoders

As in the case of the de-noising autoencoder, the hidden representation of the contractive autoencoder is often overcomplete, because the number of hidden units is greater than the number of input units. A contractive autoencoder is a heavily regularized encoder in which we do not want the hidden representation to change very significantly with small changes in input values. Obviously, this will also result in an output that is less sensitive to the input.

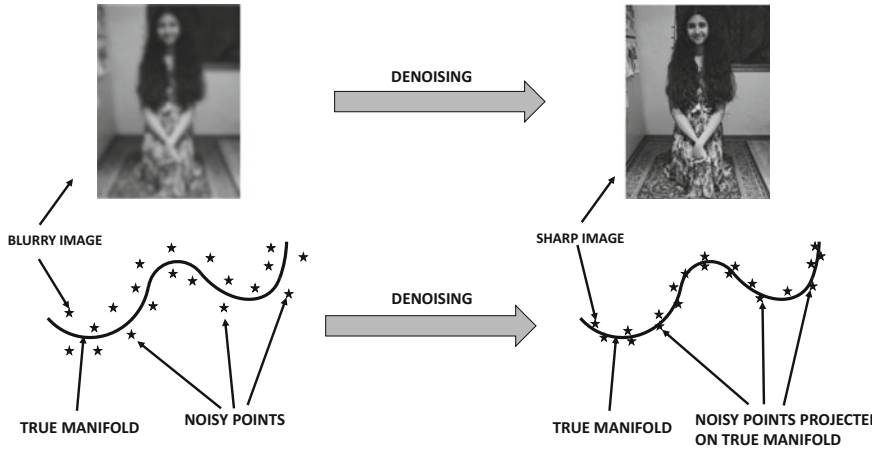


Figure 5.10: The de-noising autoencoder

Trying to create an autoencoder in which the output is less sensitive to changes in the input seems like an odd goal at first sight. After all, an autoencoder is supposed to reconstruct the data exactly. Therefore, the goals of regularization seem to be completely at odds with those of the contractive regularization portion of the loss function.

A key point is that contractive encoders are designed to be robust only to *small* changes in the input data. Furthermore, they tend to be insensitive to those changes that are inconsistent with the manifold structure of the data. In other words, if one makes a small change to the input that does not lie on the manifold structure of the input data, the contractive autoencoder will tend to damp the change in the reconstructed representation. Here, it is important to understand that the vast majority of (randomly chosen) directions in high-dimensional input data (with a much lower-dimensional manifold) tend to be approximately orthogonal to the manifold structure, which has the effect of changing the components of the change on the manifold structure. The damping of the changes in the reconstructive representation based on the local manifold structure is also referred to as the *contractive* property of the autoencoder. As a result, contractive autoencoders tend to remove noise from the input data (like de-noising autoencoders), although the mechanism for doing this is different from that of de-noising autoencoders. As we will see later, contractive autoencoders penalize the gradients of the hidden values with respect to the inputs. When the hidden values have low gradients with respect to the inputs, it means that they are not very sensitive to small changes in the inputs (although larger changes or changes parallel to manifold structure will tend to change the gradients).

For ease in discussion, we will discuss the case where the contractive autoencoder has a single hidden layer. The generalization to multiple hidden layers is straightforward. Let $h_1 \ldots h_k$ be the values of the k hidden units for the input variables $x_1 \ldots x_d$. Let the reconstructed values in the output layer be given by $\hat{x}_1 \ldots \hat{x}_d$. Then, the objective function is given by the weighted sum of the reconstruction loss and the regularization term. The loss L for a single training instance is given by the following:

$$L = \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 \tag{5.14}$$

The regularization term is constructed by using the sum of the squares of the partial derivatives of all hidden variables with respect to all input dimensions. For a problem with k hidden units denoted by $h_1 \ldots h_k$, the regularization term R can be written as follows:

$$R = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{k} \left(\frac{\partial h_j}{\partial x_i} \right)^2 \tag{5.15}$$

In the original paper [414], the sigmoid nonlinearity is used in the hidden layer, in which case the following can be shown (cf. section 2.4.2 of Chapter 2):

$$\frac{\partial h_j}{\partial x_i} = w_{ij} h_j (1 - h_j) \ \forall i, j \tag{5.16}$$

Here, w_{ij} is the weight of the input unit i to the hidden unit j.

The overall objective function for a single training instance is given by a weighted sum of the loss and the regularization terms.

$$\begin{aligned} J &= L + \lambda \cdot R \\ &= \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 + \frac{\lambda}{2} \sum_{j=1}^{k} h_j^2 (1 - h_j)^2 \sum_{i=1}^{d} w_{ij}^2 \end{aligned}$$

This objective function contains a combination of weight and hidden unit regularization. Penalties on hidden units can be handled in the same way as discussed in section 2.6.5 of Chapter 2. Let a_{h_j} be the pre-activation value for the node h_j. The backpropagation updates are traditionally defined in terms of the preactivation values, where the value of $\frac{\partial J}{\partial a_{h_j}}$ is propagated backwards. After $\frac{\partial J}{\partial a_{h_j}}$ is computed using the dynamic programming update of backpropagation from the output layer, one can further update it to incorporate the effect of hidden-layer regularization of h_j:

$$\begin{aligned} \frac{\partial J}{\partial a_{h_j}} &\Leftarrow \frac{\partial J}{\partial a_{h_j}} + \frac{\lambda}{2} \frac{\partial [h_j^2 (1 - h_j)^2]}{\partial a_{h_j}} \sum_{i=1}^{d} w_{ij}^2 \\ &= \frac{\partial J}{\partial a_{h_j}} + \lambda h_j (1 - h_j)(1 - 2h_j) \underbrace{\frac{\partial h_j}{\partial a_{h_j}}}_{h_j (1 - h_j)} \sum_{i=1}^{d} w_{ij}^2 \\ &= \frac{\partial J}{\partial a_{h_j}} + \lambda h_j^2 (1 - h_j)^2 (1 - 2h_j) \sum_{i=1}^{d} w_{ij}^2 \end{aligned}$$

The value of $\frac{\partial h_j}{\partial a_{h_j}}$ is set to $h_j(1 - h_j)$ because the sigmoid activation is assumed, although it would be different for other activations. According to the chain rule, the value of $\frac{\partial J}{\partial a_{h_j}}$ should be multiplied with the value of $\frac{\partial a_{h_j}}{\partial w_{ij}} = x_i$ to obtain the gradient of the loss with respect to w_{ij}. However, according to the *multivariable* chain rule, we also need to directly add the derivative of the regularizer with respect to w_{ij} in order to obtain the full gradient. Therefore, the partial derivative of the hidden-layer regularizer R with respect to the weight is added as follows:

$$\frac{\partial J}{\partial w_{ij}} \Leftarrow \frac{\partial J}{\partial a_{h_j}} \frac{\partial a_{h_j}}{\partial w_{ij}} + \lambda \frac{\partial R}{\partial w_{ij}}$$
$$= x_i \frac{\partial J}{\partial a_{h_j}} + \lambda w_{ij} h_j^2 (1 - h_j)^2$$

Interestingly, if a linear hidden unit is used instead of the sigmoid, it is easy to see that the objective function will become identical to that of an L_2-regularized autoencoder. Therefore, it makes sense to use this approach only with a nonlinear hidden layer, because a linear hidden layer can be handled in a much simpler way. The weights in the encoder and decoder can be either tied or independent. If the weights are tied then the gradients over both copies of a weight need to be added. The above discussion assumes a single hidden layer, although it is easy to generalize to more hidden layers. The work in [414] showed that better compression can be achieved with the use of deeper variants of the approach.

Some interesting relationships exist between the de-noising autoencoder and the contractive autoencoder. The de-noising autoencoder achieves its goals of robustness stochastically by explicitly adding noise, whereas a contractive autoencoder achieves its goals analytically by adding a regularization term. Adding a small amount of Gaussian noise in a de-noising autoencoder achieves roughly similar goals as a contractive autoencoder, when the hidden layer uses linear activation. When the hidden layer uses linear activation, the partial derivative of the hidden unit with respect to an input is simply the connecting weight, and therefore the objective function of the contractive autoencoder becomes the following:

$$J_{linear} = \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 + \frac{\lambda}{2} \sum_{i=1}^{d} \sum_{j=1}^{k} w_{ij}^2 \tag{5.17}$$

In that case, both the contractive and the de-noising autoencoders become similar to regularized singular value decomposition with L_2-regularization. The difference between the de-noising autoencoder and the contractive autoencoder is visually illustrated in Figure 5.11. In the case of the de-noising autoencoder on the left, the autoencoder learns the directions along the true manifold of uncorrupted data by using the relationship between the corrupted data in the output and the true data in the input. This goal is achieved analytically in the contractive autoencoder, because the vast majority of random perturbations are roughly orthogonal to the manifold when the dimensionality of the manifold is much smaller than the input data dimensionality. In such a case, perturbing the data point slightly does not change the hidden representation along the manifold very much. Penalizing the partial derivative of the hidden layer equally along all directions ensures that the partial derivative is significant only along the small number of directions along the true manifold, and the partial derivatives along the vast majority of orthogonal directions are close to 0. In other words, the variations that are not meaningful to the distribution of the specific training data set at hand are damped, and only the meaningful variations are kept.

Another difference between the two methods is that the de-noising autoencoder shares the responsibility for regularization between the encoder and decoder, whereas the contractive autoencoder places this responsibility only on the encoder. Only the encoder portion is used in feature extraction; therefore, contractive autoencoders are more useful for feature engineering.

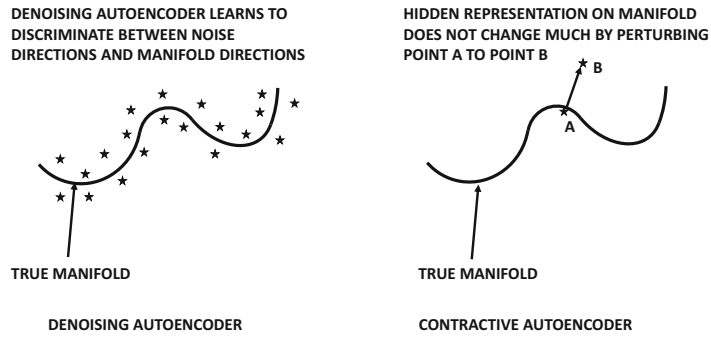
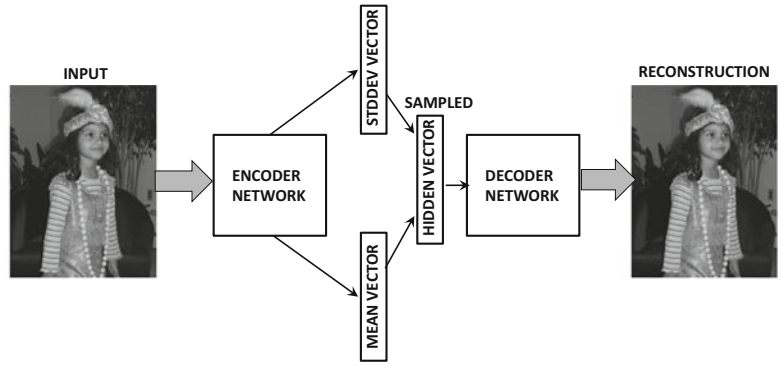


Figure 5.11: The difference between the de-noising and the contractive autoencoder

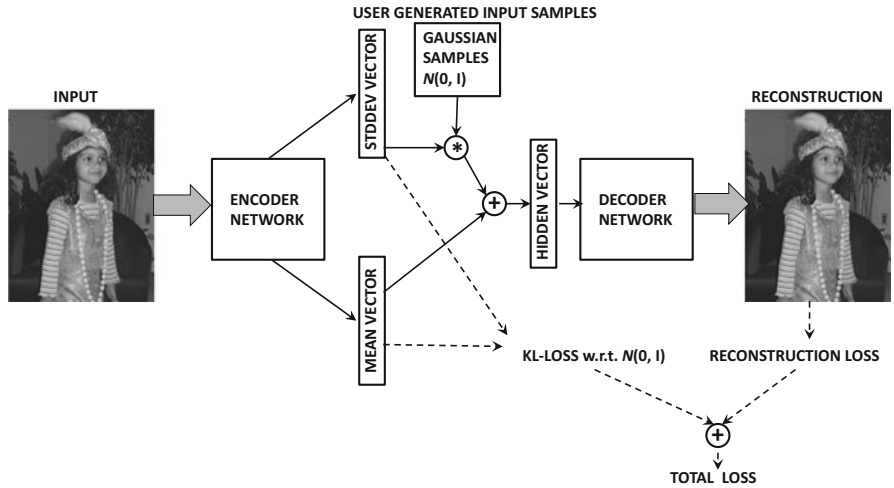
5.10.4 Hidden Probabilistic Structure: Variational Autoencoders

Just as sparse encoders impose a sparsity constraint on the hidden units, variational encoders impose a specific probabilistic structure on the hidden representation. Specifically, it is assumed that the hidden representation over the whole data should be drawn from the standard Gaussian distribution with zero mean and unit variance in each direction. By imposing this type of probabilistic structure on the hidden representation, the decoder becomes particularly useful for data generation — one can simply feed samples from the standard normal distribution to the decoder in order to generate very realistic samples of the training data. This is not possible in a traditional autoencoder, where the hidden space has arbitrary structure and a carelessly chosen sample from an unpopulated region of the hidden space might not decode into a realistic sample of the original data. The normal distribution assumption can be viewed as a *probabilistic prior*, although the *conditional* (or *posterior*) distribution of the activations in the hidden layer (after having seen a specific input object) would be a Gaussian with a different-from-zero mean and smaller-than-unit standard deviation (owing to greater knowledge of the object).

How does one control the hidden distribution with an appropriate loss function? A regularization term is used to pull the conditional Gaussian distribution (with respect to a single object) towards the standard normal distribution, which also has the effect of pulling the hidden representation of the whole data towards the standard normal distribution. The key is to use a re-parametrization approach in which the encoder creates the k-dimensional mean and standard deviations vector of the conditional Gaussian distribution and penalizes the KL-divergence of this conditional distribution from the standard normal distribution — the hidden vector for the specific object is sampled from this conditional Gaussian distribution as shown in Figure 5.12(a) in order to create its reconstruction using the decoder (whose loss is also penalized). Unfortunately, this network has a sampling component. The weights of such a network cannot be learned by backpropagation because the stochastic portions of the computations are not differentiable, and therefore backpropagation cannot be not used. Therefore, the stochastic part of it can be addressed by the user explicitly generating k-dimensional samples in which each component is drawn from the standard normal distribution. The mean and standard deviation output by the encoder are used to scale and translate the input sample from the Gaussian distribution. This architecture is shown in Figure 5.12(b). By generating the stochastic potion explicitly as a part of the input, the resulting architecture is now fully deterministic, and its weights can be learned by backpropagation. It is noteworthy that the backpropagation updates will depend on the



(a) Point-specific Gaussian distribution (stochastic and non-differentiable loss)



(b) Point-specific Gaussian distribution (deterministic and differentiable loss)

Figure 5.12: Re-parameterizing a variational autoencoder

stochastically sampled noise (although the effects of this noise will be muted over a large number of samples).

For each object $\overline{X}$, separate hidden activations for the mean and standard deviation are created by the encoder. The k-dimensional activations for the mean and standard deviation are denoted by $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$, respectively. In addition, a k-dimensional sample $\overline{z}$ is generated from $\mathcal{N}(0, I)$, where I is the identity matrix, and treated as an input into the hidden layer by the user. The hidden representation $\overline{h}(\overline{X})$ is created by scaling this random input vector $\overline{z}$ with the mean and standard deviation as follows:

$$\overline{h}(\overline{X}) = \overline{z} \odot \overline{\sigma}(\overline{X}) + \overline{\mu}(\overline{X}) \tag{5.18}$$

Here, $\odot$ indicates element-wise multiplication. These operations are shown in Figure 5.12(b) with the little circles containing the multiplication and addition operators. The elements of the vector $\overline{h}(\overline{X})$ for a particular object will obviously diverge from the standard normal distribution unless the vectors $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$ contain only 0s and 1s, respectively. This will not be the case because of the reconstruction component of the loss, which forces the

conditional distributions of the hidden representations of particular points to have different means and lower standard deviations than that of the standard normal distribution (which is like a prior distribution). The distribution of the hidden representation of a particular point is a posterior distribution (conditional on the specific training data point), and therefore it will differ from the Gaussian prior. The overall loss function is expressed as a weighted sum of the reconstruction loss and the regularization loss. One can use a variety of choices for the reconstruction error, and for simplicity we will use the squared loss, which is defined as follows:

$$L = ||\overline{X} - \overline{X}'||^2 \tag{5.19}$$

Here, $\overline{X}'$ is the reconstruction of the input point $\overline{X}$ from the decoder. The regularization loss R is simply the Kullback-Leibler (KL)-divergence measure of the conditional hidden distribution with parameters $(\overline{\mu}(\overline{X}), \overline{\sigma}(\overline{X}))$ with respect to the k-dimensional spherical Gaussian distribution with unit variance in each direction. The KL-divergence term is defined as follows:

$$R = \frac{1}{2}\left(\underbrace{||\overline{\mu}(\overline{X})||^2}_{\overline{\mu}(\overline{X})_i \Rightarrow 0} + \underbrace{||\overline{\sigma}(\overline{X})||^2 - 2\sum_{i=1}^{k} \ln(\overline{\sigma}(\overline{X})_i) - k}_{\overline{\sigma}(\overline{X})_i \Rightarrow 1} \right) \tag{5.20}$$

The overall objective function J for the data point $\overline{X}$ is defined as the weighted sum of the reconstruction loss and the regularization term:

$$J = L + \lambda R \tag{5.21}$$

Here, $\lambda > 0$ is the regularization parameter. Small values of λ will favor exact reconstruction like a traditional autoencoder. The regularization term takes on its minimum value of 0, when $(\overline{\mu}(\overline{X}), \overline{\sigma}(\overline{X})) = (\overline{0}, \overline{1})$ corresponds to the standard Gaussian with zero mean and unit variance. However, the reconstruction portion of the objective function will push $\overline{\mu}(\overline{X})$ towards non-zero values favoring faithful reconstruction of $\overline{X}$ and the components of $\overline{\sigma}(\overline{X})$ to values smaller than 1 (favoring the addition of less point-specific noise to the hidden representation before reconstruction). Even though the *point-specific* posteriors represented by $(\overline{\mu}(\overline{X}), \overline{\sigma}(\overline{X}))$ will be far from the standard Gaussian, the distribution of the hidden representation *over all training points* will be much closer to the standard Gaussian because of the regularization term.

The regularization term forces the hidden representations to be stochastic, so that multiple hidden representations generate almost the same point. This increases generalization power because it is easier to model a new image that is like (but not an exact likeness of) an image in the training data within a stochastic range of hidden values. However, since there will be overlaps among the distributions of the hidden representations of similar points, it has some undesirable side effects. For example, the reconstructions of image data points tend to be blurry. This is caused by an averaging effect over somewhat similar images. In the extreme case, if the value of λ is chosen to be exceedingly large, all points will have the same hidden (posterior) distribution — the isotropic Gaussian with zero mean and unit variance. The resulting reconstruction results will exhibit a gross averaging effect over the entire training data, which will obviously not be meaningful.

Training the Variational Autoencoder

The training of a variational autoencoder is relatively straightforward because the stochasticity has been pulled out as an additional input. One can backpropagate as in any traditional neural network. The only difference is that one needs to backpropagate across the unusual form of Equation 5.18. Furthermore, one needs to account for the penalties of the hidden layer during backpropagation.

First, one can backpropagate the loss L up to the hidden state $\overline{h}(\overline{X}) = (h_1 \ldots h_k)$ using traditional methods. Let $\overline{z} = (z_1 \ldots z_k)$ be the k random samples from $\mathcal{N}(0, 1)$, which are used in the current iteration. In order to backpropagate from $\overline{h}(\overline{X})$ to $\overline{\mu}(\overline{X}) = (\mu_1 \ldots \mu_k)$ and $\overline{\sigma}(\overline{X}) = (\sigma_1 \ldots \sigma_k)$, one can use the following relationship:

$$J = L + \lambda R \tag{5.22}$$

$$\frac{\partial J}{\partial \mu_i} = \frac{\partial L}{\partial h_i} \underbrace{\frac{\partial h_i}{\partial \mu_i}}_{=1} + \lambda \frac{\partial R}{\partial \mu_i} \tag{5.23}$$

$$\frac{\partial J}{\partial \sigma_i} = \frac{\partial L}{\partial h_i} \underbrace{\frac{\partial h_i}{\partial \sigma_i}}_{=z_i} + \lambda \frac{\partial R}{\partial \sigma_i} \tag{5.24}$$

The values below the under-braces show the evaluations of partial derivatives of h_i with respect to μ_i and σ_i, respectively. Note that the values of $\frac{\partial h_i}{\partial \mu_i} = 1$ and $\frac{\partial h_i}{\partial \sigma_i} = z_i$ are obtained by differentiating Equation 5.18 with respect to μ_i and σ_i, respectively. The value of $\frac{\partial L}{\partial h_i}$ on the right-hand side is available from backpropagation. The values of $\frac{\partial R}{\partial \mu_i}$ and $\frac{\partial R}{\partial \sigma_i}$ are straightforward derivatives of the KL-divergence in Equation 5.20. Subsequent error propagation from the activations for $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$ can proceed in a similar way to the normal workings of the backpropagation algorithm.

The architecture of the variational autoencoder is considered fundamentally different from other types of autoencoders because it models the hidden variables in a stochastic way. However, there are still some interesting connections. In the de-noising autoencoder, one adds noise to the input; however, there is no constraint on the shape of the hidden distribution. In the variational autoencoder, one works with a stochastic hidden representation, although the stochasticity is pulled out by using it as an additional input during training. In other words, noise is added to the hidden representation rather than the input data. The variational approach improves generalization, because it encourages each input to map to its own stochastic region in the hidden space rather than mapping it to a single point. Small changes in the hidden representation, therefore, do not change the reconstruction too much. This assertion would also be true with a contractive autoencoder. However, constraining the shape of the hidden distribution to be Gaussian is a more fundamental difference of the variational autoencoder from other types of transformations.

5.10.4.1 Reconstruction and Generative Sampling

The approach can be used for creating the reduced representations as well as generating samples. In the case of data reduction, a Gaussian distribution with mean $\overline{\mu}(\overline{X})$ and standard deviation $\overline{\sigma}(\overline{X})$ is obtained, which represents the distribution of the hidden representation.

However, a particularly interesting application of the variational autoencoder is to generate samples from the underlying data distribution. Just as feature engineering methods

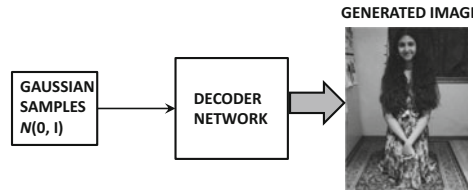


Figure 5.13: Generating samples from the variational autoencoder. The images are illustrative only.

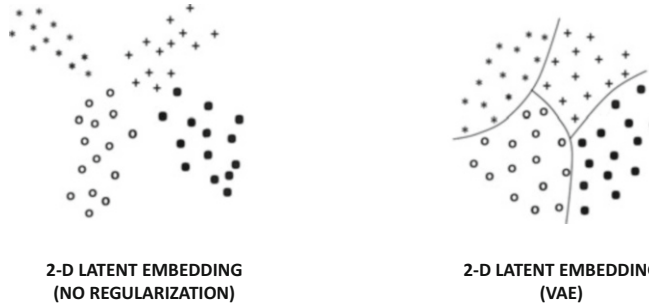


Figure 5.14: Illustrations of the embeddings created by a variational autoencoder in relation to the unregularized version. The unregularized version has large discontinuities in the latent space, which might not correspond to meaningful points. The Gaussian embedding of the points in the variational autoencoder makes sampling possible.

use only the encoder portion of the autoencoder (once training is done), variational autoencoders use only the decoder portion. The basic idea is to repeatedly draw a point from the Gaussian distribution and feed it to the hidden units in the decoder. The resulting "reconstruction" output of the decoder will be a point satisfying a similar distribution as the original data. As a result, the generated point will be a realistic sample from the original data. The architecture for sample generation is shown in Figure 5.13. The shown image is illustrative only, and does not reflect the actual output of a variational autoencoder (which is generally of somewhat lower quality). To understand why a variational autoencoder can generate images in this way, it is helpful to view the typical types of embeddings an unregularized autoencoder would create versus a method like the variational autoencoder. In the left side of Figure 5.14, we have shown an example of the 2-dimensional embeddings of the training data created by an unregularized autoencoder of a four-class distribution (e.g., four digits of MNIST). It is evident that there are large discontinuities in particular regions of the latent space, and that these sparse regions may not correspond to meaningful points. On the other hand, the regularization term in the variational autoencoder encourages the training points to be (roughly) distributed in a Gaussian distribution, and there are far fewer discontinuities in the embedding on the right-hand side of Figure 5.14. Consequently, sampling from any point in the latent space will yield meaningful reconstructions of one of the four classes (i.e., one of the digits of MNIST). Furthermore, "walking" from one point in the latent space to another along a straight line in the second case will result in a smooth transformation across classes. For example, walking from a region containing instances of '4' to a region containing instances of '7' in the latent space of the MNIST data set would result in a slow change in the style of the digit '4' until a transition point, where the handwritten digit could be interpreted either as a '4' or a '7'. This situation does occur in real

settings as well because such types of confusing handwritten digits do occur in the MNIST data set. Furthermore, the placement of different digits within the embedding would be such that digit pairs with smooth transitions at confusion points (e.g., [4, 7] or [5, 6]) are placed adjacent to one another in the latent space.

It is important to understand that the generated objects are often similar to but not exactly the same as those drawn from the training data. Because of its stochastic nature, the variational autoencoder has the ability to explore different modes of the generation process, which leads to a certain level of creativity in the face of ambiguity. This property can be put to good use by conditioning the approach on another object.

5.10.4.2 Conditional Variational Autoencoders

One can apply conditioning to variational autoencoders in order to obtain some interesting results [479, 530]. The basic idea in conditional variational autoencoders is to add an additional conditional input, which typically provides a related context. For example, the context might be a damaged image with missing holes, and the job of the autoencoder is to reconstruct it. Predictive models will generally perform poorly in this type of setting because the level of ambiguity may be too large, and an averaged reconstruction across all images might not be useful. During the training phase, pairs of damaged and original images are needed, and therefore the encoder and decoder are able to learn how the context relates to the images being generated from the training data. The architecture of the training phase is illustrated in the upper part of Figure 5.15. The training is otherwise similar to the unconditional variational autoencoder. During the testing phase, the context is provided as an additional input, and the autoencoder reconstructs the missing portions in a reasonable way based on the model learned in the training phase. The architecture of the reconstruction phase is illustrated in the lower part of Figure 5.15. The simplicity of this architecture is particularly notable. The shown images are only illustrative; in actual executions on image data, the generated images are often blurry, especially in the missing portions. This is a type of image-to-image translation approach, which will be revisited in Chapter 12 under the context of a discussion on *generative adversarial networks*.

5.10.4.3 Relationship with Generative Adversarial Networks

Variational autoencoders are closely related to another class of models, referred to as generative adversarial networks. However, there are some key differences as well. Like variational autoencoders, generative adversarial networks can be used to create images that are similar to a base training data set. Furthermore, conditional variants of both models are useful for completing missing data, especially in cases where the ambiguity is large enough to require a certain level of creativity from the generative process. However, the results of generative adversarial networks are often more realistic because the decoders are explicitly trained to create good counterfeits. This is achieved by having a discriminator as a judge of the quality of the generated objects. Furthermore, the objects are also generated in a more creative way because the generator is never shown the original objects in the training data set, but is only given guidance to fool the discriminator. As a result, generative adversarial networks learn to create creative counterfeits. In certain domains such as image and video data, this approach can have remarkable results; unlike variational autoencoders, the quality of the images is not blurry. One can create vivid images and videos with an artistic flavor, that give the impression of dreaming. These techniques can also be used in numerous applications like text-to-image or image-to-image translation. For example, one can specify a text

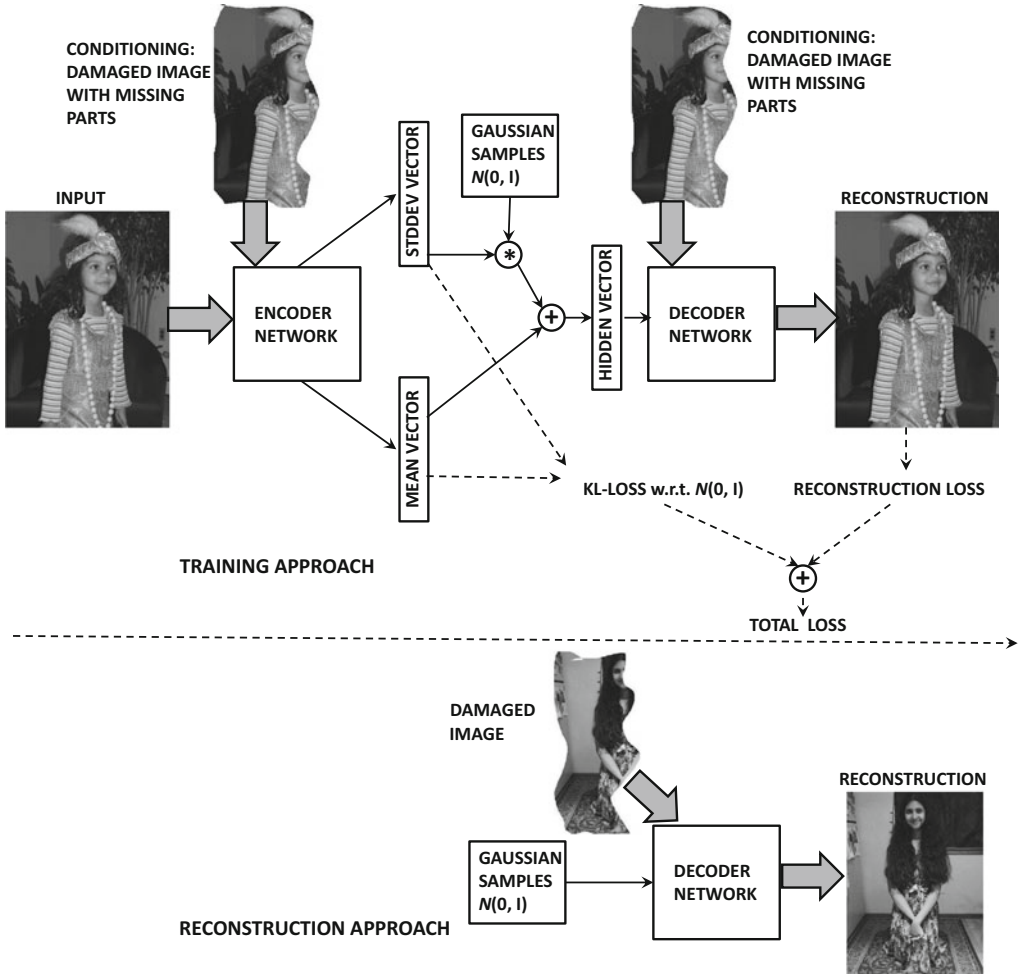


Figure 5.15: Reconstructing damaged images with the conditional variational autoencoder. The images are illustrative only.

description, and then obtain a fantasy image that matches the description [408]. Generative adversarial networks are discussed in section 12.5 of Chapter 12.

5.11 Summary

Large neural networks often face the challenge of overfitting. Reducing network size up front results in suboptimal solutions when the underlying model is complex. A more flexible approach is to use tunable regularization, in which a large number of parameters are allowed. In such cases, the regularization restricts the size of the parameter space in a soft way. The most common form of regularization is penalty-based regularization on either the parameters or the hidden units. Ensemble learning is a common approach to reduce variance, and some ensemble methods like *Dropout* are specifically designed for neural networks. Other common regularization methods include early stopping and pretraining. Pretraining acts as

a regularizer by acting as a form of semi-supervised learning, which works from the simple to the complex by initializing with a simple heuristic and using backpropagation to discover refined solutions. Other related techniques include curriculum and continuation methods, which also work from the simple to the complex in order to provide solutions with low generalization error. Although overfitting is a less serious problem in unsupervised settings, regularization is often used to impose structure on the learned representations.

5.12 Bibliographic Notes and Software Resources

A detailed discussion of the bias-variance trade-off may be found in [187]. The bias-variance trade-off originated in the field of statistics, where it was proposed in the context of the regression problem. The generalization to the case of binary loss functions in classification was proposed in [257, 260]. Early methods for reducing overfitting were proposed in [185, 288] in which unimportant weights were removed from a network to reduce its parameter footprint. It was shown that this type of pruning had significant benefits in terms of generalization. The early work also showed [470] that deep and narrow networks tended to generalize better than broad and shallow networks. This is primarily because the depth imposes a structure on the data, and can represent the data in a fewer number of parameters. A recent study of model generalization in neural networks is provided in [582].

The use of L_2-regularization in regression dates back to Tikhonov-Arsenin's seminal work [516]. The equivalence of Tikhonov regularization and training with noise was shown by Bishop [43]. The use of L_1-regularization is studied in detail in [189]. Several regularization methods have also been proposed that are specifically designed for neural architectures. For example, the work in [211] proposes a regularization technique that constrains the norm of each layer in a neural network. Sparse representations of the data are explored in [69, 279, 280, 290, 365].

Detailed discussions of ensemble methods for classification may be found in [458, 591]. The bagging and subsampling methods are discussed in [49, 56]. The work in [535] proposes an ensemble architecture that is inspired by a random forest. This architecture is illustrated in Figure 2.10 of Chapter 1. This type of ensemble is particularly well suited for problems with small data sets, where a random forest is known to work well. The approach for random edge dropping was introduced in the context of outlier detection [64], whereas the *Dropout* approach was presented in [483]. The work in [592] discusses the notion that it is better to combine the results of the top-performing ensemble components rather than combining all of them. Most ensemble methods are designed for variance reduction, although a few techniques like *boosting* [127] are also designed for bias reduction. Boosting has also been used in the context of neural network learning [455]. However, the use of boosting in neural networks is generally restricted to the incremental addition of hidden units based on error characteristics. A key point about boosting is that it tends to overfit the data, and is therefore suitable for high-bias learners but not high-variance learners. Neural networks are inherently high-variance learners. The relationship between boosting and certain types of neural architectures is pointed out in [32]. Combinations of data perturbation methods with ensembles are discussed in [5]. Ensemble methods for neural networks are proposed in [180].

Different types of pretraining have been explored in the context of neural networks [31, 116, 206, 402, 526]. The earliest methods for unsupervised pretraining were proposed in [206]. The original work of pretraining [206] was based on probabilistic graphical models (cf. section 7.7) and was later extended to conventional autoencoders [402, 526].

Compared to unsupervised pretraining, the effect of supervised pretraining is limited [31]. A detailed discussion of why unsupervised pretraining helps deep learning is provided in [116]. This work posits that unsupervised pretraining implicitly acts as a regularizer, and therefore it improves the generalization power to unseen test instances. This fact is also evidenced by the experimental results in [31], which show that supervised variations of pretraining do not help as much as unsupervised variations of pretraining. Pretraining also does not seem to help with certain types of tasks [312]. Another form of semi-supervised learning can be performed with *ladder networks* [404, 520], in which skip-connections are used in conjunction with an autoencoder-like architecture.

Curriculum and continuation learning are applications of the principle of moving from simple to complex models. Continuation learning methods are discussed in [350, 557]. A number of methods were proposed in the early years [114, 439, 480] that showed the advantages of curriculum learning. The basic principles of curriculum learning are discussed in [248]. The relationship between curriculum and continuation learning is explored in [33].

Sparse autoencoders are discussed in [69, 279, 280, 290, 365]. De-noising autoencoders are discussed in [526]. The contractive autoencoder is discussed in [414]. The use of de-noising autoencoders in recommender systems is discussed in [488, 555]. The ideas in the contractive autoencoder are reminiscent of *double backpropagation* [109] in which small changes in the input are not allowed to change the output. Related ideas are also discussed in the *tangent classifier* [415].

The variational autoencoder was introduced in [252, 416]. The use of importance weighting to improve over the representations learned by the variational autoencoder is discussed in [58]. Conditional variational autoencoders are discussed in [479, 530]. A tutorial on variational autoencoders is found in [108]. Generative variants of de-noising autoencoders are discussed in [34]. Variational autoencoders are closely related to generative adversarial networks, which are discussed in Chapter 12. Closely related methods for designing adversarial autoencoders are discussed in [323].

Software Resources

Numerous ensemble methods are available from machine learning libraries like *scikit-learn* [611]. Most of the weight-decay and penalty-based methods are available as standardized options in the deep learning libraries. However, techniques like *Dropout* are application-specific and need to be implemented from scratch. Implementations of several different types of autoencoders may be found in [618]. Several implementations of the variational autoencoder may be found in [619, 620, 662].

5.13 Exercises

1. Consider two neural networks used for regression modeling, each with an input layer, 10 hidden layers containing 100 units each, and a single linear output unit. The only difference is that one of them uses linear activations in the hidden layers and the other uses sigmoid activations. Which model will have higher variance of prediction?

2. Consider a situation in which you have four attributes $x_1 \ldots x_4$, and the dependent variable y is such that $y = 2x_1$. Create a tiny training data set of 5 distinct examples in which a linear regression model without regularization will have an infinite number of coefficient solutions with $w_1 = 0$. Discuss the performance of such a model on out-of-sample data. Why will regularization help?

3. Implement a perceptron with and without regularization. Test the accuracy of both variations on the training data and the out-of-sample data on the *Ionosphere* data set of the *UCI Machine Learning Repository* [625]. What is the effect of regularization in the two cases? Repeat the experiment with smaller samples of the *Ionosphere* training data, and report your observations.

4. Implement an autoencoder with a single hidden layer. Reconstruct inputs for the *Ionosphere* data set of the previous exercise with (a) no added noise and weight regularization, (b) added Gaussian noise and no weight regularization.

5. The discussion in the chapter uses an example of sigmoid activation for the contractive autoencoder. Consider a contractive autoencoder with a single hidden layer and ReLU activation. Discuss how the updates change when ReLU activation is used.

6. Suppose that you have a model that provides around 80% accuracy on the training as well as on the out-of-sample test data. Would you recommend increasing the amount of data or adjusting the model to improve accuracy?

7. In the chapter, we showed that adding Gaussian noise to the input features in linear regression is equivalent to L_2-regularization of linear regression. Discuss why adding of Gaussian noise to the input data in a de-noising single-hidden layer autoencoder with linear units is roughly equivalent to L_2-regularized singular value decomposition.

8. Consider a network with a single input layer, two hidden layers, and a single output predicting a binary label. All hidden layers use the sigmoid activation function and no regularization is used. The input layer contains d units, and each hidden layer contains p units. Suppose that you add an additional hidden layer between the two current hidden layers, and this additional hidden layer contains q linear units.

(a) Even though the number of parameters have increased by adding the hidden layer, discuss why the capacity of this model will decrease when $q < p$.

(b) Does the capacity of the model increase when $q > p$?

9. Bob divided the labeled classification data into a portion used for model construction and another portion for validation. Bob then tested 1000 neural architectures by learning parameters (backpropagating) on the model-construction portion and testing its accuracy on the validation portion. Discuss why the resulting model is likely to yield poorer accuracy on the out-of-sample test data as compared to the validation data, even though the validation data was not used for learning parameters. Do you have any recommendations for Bob on using the results of his 1000 validations?

10. Does the classification accuracy on the training data generally improve with increasing training data size? How about the point-wise average of the loss on training instances? For what types of data sizes do training and testing accuracy become similar? Explain your answer.

11. What is the effect of increasing the regularization parameter on the training and testing accuracy? For what types of regularization parameters do training and testing accuracy become similar?

12. Consider an autoencoder with k real-valued units $h_1 \ldots h_k$ in the hidden layer. In addition to the least-squares loss L, you apply the following regularization term R in order to create the objective function $L + \lambda R$ for regularization parameter $\lambda > 0$:

$$R = \sum_{i=1}^{k} \max\{1 - h_i, 0\}$$

Discuss the effect of this regularization on the hidden representation. [Hint: Hinge Loss]

Chapter 6

Radial Basis Function Networks

"Two birds disputed about a kernel, when a third swooped down and carried it off."– African Proverb

6.1 Introduction

Radial basis function (RBF) networks represent a fundamentally different architecture from what we have seen in ?the previous chapters. All the previous chapters use a feed-forward network in which the inputs are transmitted forward from layer to layer in a similar fashion in order to create the final outputs. A feed-forward network might have many layers, and the nonlinearity is typically created by the repeated composition of activation functions. On the other hand, an RBF network typically uses only an input layer, a single hidden layer (with a special type of behavior defined by RBF functions), and an output layer. Although it is possible to replace the output layer with multiple feed-forward layers (like a conventional network), the resulting network is still quite shallow, and its behavior is strongly influenced by the nature of the special hidden layer. For simplicity in discussion, we will work with only a single output layer. As in feed-forward networks, the input layer is not really a computational layer, and it only carries the inputs forward. The nature of the computations in the hidden layer are very different from what we have seen so far in feed-forward networks. In particular, the hidden layer performs a computation based on a comparison with a *prototype vector*, which has no exact counterpart in feed-forward networks. The structure and the computations performed by the special hidden layer is the key to the power of the RBF network.

One can characterize the difference in the functionality of the hidden and output layers as follows:

1. The hidden layer takes the input points, in which the class structure might not be linearly separable, and transforms them into a new space that is (often) linearly separable. The hidden layer often has higher dimensionality than the input layer, because

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_6

transformation to a higher-dimensional space is often required in order to ensure linear separability. This principle is based on *Cover's theorem on separability of patterns* [87], which states that pattern classification problems are more likely to be linearly separable when cast into a high-dimensional space with a nonlinear transformation. Furthermore, certain types of transformations in which features represent small localities in the space are more likely to lead to linear separability. Although the dimensionality of the hidden layer is typically greater than the input dimensionality, it is always less than or equal to the number of training points. An extreme case in which the dimensionality of the hidden layer is equal to the number of training points can be shown to be roughly equivalent to *kernel learners*. Examples of such models include kernel regression and kernel support vector machines.

2. The output layer uses an activation and loss function based on nature of the application. For example, binary predictions might use logistic loss, whereas real-valued predictions might used squared loss with identity activation. The connections from the hidden to the output layer have weights attached to them. The computations in the output layer are performed in the same way as in a standard feed-forward network. Although it is also possible to replace the output layer with multiple feed-forward layers, this is uncommon in practice.

Special cases of the RBF network can be used to implement kernel regression, least-squares kernel classification, and the kernel support-vector machine. Like feed-forward networks, RBF networks are universal function approximators.

The layers of the RBF network are designed as follows:

1. The input layer simply transmits from the input features to the hidden layers. Therefore, the number of input units is exactly equal to the dimensionality d of the data. As in the case of feed-forward networks, no computation is performed in the input layers. As in all feed-forward networks, the input units are fully connected to the hidden units and carry their input forward.

2. The computations in the hidden layers are based on comparisons with *prototype vectors*. Each hidden unit contains a d-dimensional prototype vector. Let the prototype vector of the ith hidden unit be denoted by $\overline{\mu}_i$. In addition, the ith hidden unit contains a bandwidth denoted by σ_i. Although the prototype vectors are always specific to particular units, the bandwidths of different units σ_i are often set to the same value σ. The prototype vectors and bandwidth(s) are usually learned either in an unsupervised way, or with the use of mild supervision.

   Then, for any input training point $\overline{X}$, the activation $\Phi_i(\overline{X})$ of the ith hidden unit is defined as follows:

$$h_i = \Phi_i(\overline{X}) = \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right) \quad \forall i \in \{1, \ldots, m\} \tag{6.1}$$

   The total number of hidden units is denoted by m. Each of these m units is designed to have a high level of influence on the particular cluster of points that is closest to its prototype vector. Therefore, one can view m as the number of clusters used for modeling, and it represents an important hyper-parameter available to the algorithm. For low-dimensional inputs, it is typical for the value of m to be larger than the input dimensionality d, but smaller than the number of training points n.

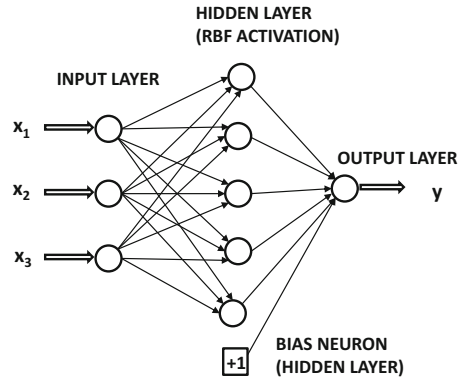


Figure 6.1: An RBF network: Note that the hidden layer is broader than the input layer, which is typical (but not mandatory).

3. For any particular training point $\overline{X}$, let h_i be the output of the ith hidden unit, as defined by Equation 6.1. The weights of the connections from the hidden to the output nodes are set to w_i. Then, the prediction $\hat{y}$ of the RBF network in the output layer is defined as follows:

$$\hat{y} = \sum_{i=1}^{m} w_i h_i = \sum_{i=1}^{m} w_i \Phi_i(\overline{X}) = \sum_{i=1}^{m} w_i \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right)$$

The variable $\hat{y}$ has a circumflex on top to indicate the fact that it is a predicted value rather than observed value. If the observed target is real-valued, then one can set up a least-squares loss function, which is much like that in a feed-forward network. The values of the weights $w_1 \ldots w_m$ need to be learned in a supervised way.

An additional detail is that the hidden layer of the neural network contains bias neurons. Note that the bias neuron can be implemented by a single hidden unit in the output layer, which is always on. One can also implement this type of neuron by creating a hidden unit in which the value of σ_i is ∞. In either case, it will be assumed throughout the discussions in this chapter that this special hidden unit is absorbed among the m hidden units. Therefore, it is not treated in any special way. An example of an RBF network is illustrated in Figure 6.1.

In the RBF network, there are two sets of computations corresponding to the hidden layer and the output layer. The parameters $\overline{\mu}_i$ of the hidden layer are learned in an unsupervised way, whereas those of the output layer are learned in a supervised way with gradient descent. The latter is similar to the case of the feed-forward network. The prototypes $\overline{\mu}_i$ may either be sampled from the data, or be set to be the m *centroids* of an m-way clustering algorithm. In other words, we can partition the training data into m clusters with an off-the-shelf clustering algorithm, and use the means of the m clusters as the m prototypes. The parameters σ_i are often set to a single hyper-parameter value σ, which is learned by holding out a small part of the data to set σ to a value that maximizes the accuracy on the held out part of the data. The different ways of training the neural network are discussed in detail in section 6.2.

An interesting special case is when the prototypes are set to the individual training points (and therefore the value m is the same as the number of training examples). In such cases, RBF networks can be shown to specialize to well-known *kernel methods* [7] in machine

learning. This relationship will be discussed in detail in section 6.4, and it suggests that RBF networks have greater power and flexibility than do kernel methods.

When to Use RBF Networks

A key point is that the hidden layer of the RBF network is created in an unsupervised way, tending to make it robust to all types of noise. Feed-forward networks can become increasingly sensitive to the effects of noise with grater depth. At the same time, there are limitations with respect to how much structure in the data an RBF network can learn. Deep feed-forward networks are effective at learning from data with a rich structure because the multiple layers of nonlinear activations allow the network to learn specific types of patterns. Furthermore, by adjusting the structure of connections, one can incorporate domain-specific insights in feed-forward networks. Examples of such settings include recurrent and convolutional neural networks. The single layer of an RBF network limits the amount of structure that one can learn. Although both RBF networks and deep feed-forward networks are known to be universal function approximators, the former works better for noisy data with limited structure, whereas the latter is suitable for data with rich structure (but somewhat less noise).

Chapter Organization

This chapter is organized as follows. The next section discusses the various training methods for RBF networks. The use of RBF networks in classification and interpolation is discussed in section 6.3. The relationship of the RBF method to kernel regression and classification is discussed in section 6.4. A summary is provided in section 6.5.

6.2 Training an RBF Network

The training of an RBF network is very different from that of a feed-forward network, which is fully integrated across different layers. In an RBF network, the training of the hidden layer is typically done in an unsupervised manner. While it is possible, in principle, to train the prototype vectors and the bandwidths using backpropagation, the problem is that there are more local minima on the loss surface of RBF networks compared to feed-forward networks. Therefore, the supervision in the hidden layer (when used) is often relatively gentle, or it is restricted only to fine-tuning weights that have already been learned. Nevertheless, since overfitting seems to be a pervasive problem with the supervised training of the hidden layer, our discussion will be restricted to unsupervised methods. In the following, we will first discuss the training of the hidden layer of an RBF network, and then discuss the training of the output layer.

6.2.1 Training the Hidden Layer

The hidden layer of the RBF network contains several parameters, including the prototype vectors $\overline{\mu}_1 \ldots \overline{\mu}_m$, and the bandwidths $\sigma_1 \ldots \sigma_m$. The hyperparameter m controls the number of hidden units. In practice, a separate value of σ_i is not set for each unit, and all units have the same bandwidth σ. However, the mean values $\overline{\mu}_i$ for the various hidden units are different because they define the all-important prototype vectors. The complexity of the model is regulated by the number of hidden units and the bandwidth. The combination of a small bandwidth and a large number of hidden units increases the model complexity,

and is a useful setting when the amount of data is large. Smaller data sets require fewer units and larger bandwidths to avoid overfitting. The value of m is typically larger than the input data dimensionality, but it is never larger than the number of training points. Setting the value of m equal to the number of training points, and using each training point as a prototype in a hidden node, makes the approach equivalent to traditional kernel methods.

The bandwidth also depends on the chosen prototype vectors $\overline{\mu}_1 \ldots \overline{\mu}_m$. Ideally, the bandwidths should be set in a way that each point should be (significantly) influenced by only a small number of prototype vectors, which correspond to its closest clusters. Setting the bandwidth too large or too small compared to the inter-prototype distance will lead to under-fitting and over-fitting, respectively. Let d_{max} be maximum distance between pairs of prototype centers, and d_{ave} be the average distance between them. Then, two heuristic ways of setting the bandwidth are as follows:

$$\sigma = \frac{d_{max}}{\sqrt{m}}$$
$$\sigma = 2 \cdot d_{ave}$$

One problem with this choice of σ is that the optimal value of the bandwidth might vary in different parts of the input space. For example, the bandwidth in a dense region in the data space should be smaller than the bandwidth in a sparse region of the space. The bandwidth should also depend on how the prototype vectors are distributed in the space. Therefore, one possible solution is to choose the bandwidth σ_i of the ith prototype vector to be equal to its distance to its rth nearest neighbor among the prototypes. Here, r is a small value like 5 or 10.

However, these are only heuristic rules. It is possible to fine-tune these values by using a held-out portion of the data set. In other words, candidate values of σ are generated in the neighborhood of the above recommended values of σ (as an initial reference point). Then, multiple models are constructed using these candidate values of σ (including the training of the output layer). The choice of σ that provides the least error on the held-out portion of the training data set is used. This type of approach does use a certain level of supervision in the selection of the bandwidth, without getting stuck in local minima. However, the nature of the supervision is quite gentle, which is particularly important when dealing with the parameters of the first layer in an RBF network.

The selection of the prototype vectors is somewhat more complex. In particular, the following choices are often made:

1. The prototype vectors can be randomly sampled from the n training points. A total of $m < n$ training points are sampled in order to create the prototype vectors. The main problem with this approach is that it will over-represent prototypes from dense regions of the data, whereas sparse regions might get few or no prototypes. As a result, the prediction accuracy in such regions will suffer.

2. A k-means clustering algorithm can be used in order to create m clusters. The centroid of each of these m clusters can be used as a prototype. The use of the k-means algorithm is the most common choice for learning the prototype vectors.

3. Variants of clustering algorithms that partition the data *space* (rather than the points) are also used. A specific example is the use of decision trees to create the prototypes.

4. An alternative method for training the hidden layer is by trying different prototypes for the units in the hidden layer. This approach uses a certain level of supervision. In

this approach, the prototype vectors are selected one by one from the training data in order to minimize the residual error of prediction on an out-of-sample test set. Since this approach requires understanding of the training of the output layer, its discussion will be deferred to a later section.

In the following, we briefly describe the k-means algorithm for creating the prototypes because it is the most common choice in real implementations. The k-means algorithm is a classical technique in the clustering literature. It uses the cluster prototypes as the prototypes for the hidden layer in the RBF method. Broadly speaking, the k-means algorithm proceeds as follows. At initialization, the m cluster prototypes are set to m random training points. Subsequently, each of the n data points is assigned to the prototype to which it has the smallest Euclidean distance. The assigned points of each prototype are averaged in order to create a new cluster center. In other words, the centroid of the created clusters is used to replace its old prototype with a new prototype. This process is repeated iteratively to convergence. Convergence is reached when the cluster assignments do not change significantly from one iteration to the next.

6.2.2 Training the Output Layer

The output layer is trained after the hidden layer has been trained. The training of the output layer is quite straightforward, because it uses only a single layer with linear activation. For ease in discussion, we will first consider the case in which the target of the output layer is real-valued. Later, we will discuss other settings. The output layer contains an m-dimensional vector of weights $\overline{W} = [w_1 \ldots w_m]^T$ that needs to be learned. Assume that the vector $\overline{W}$ is a column vector.

Consider a situation in which the training data set contains n points $\overline{X}_1 \ldots \overline{X}_n$, each of which is a row vector. The input $\overline{X}_i$ is transformed to the m-dimensional row vector $\overline{H}_i$ in the hidden layer of m units. One could stack the n row vectors $\overline{H}_1 \ldots \overline{H}_n$ on top of one another to create an $n \times m$ matrix H. Furthermore, the observed targets of the n training points are denoted by $y_1, y_2, \ldots y_n$, which can be written as the n-dimensional column vector $\overline{y} = [y_1 \ldots y_n]^T$.

The predictions of the n training points are given by the elements of the n-dimensional column vector $H\overline{W}$. Ideally, we would like these predictions to be as close to the observed vector $\overline{y}$ as possible. Therefore, the loss function L for learning the output-layer weights is obtained by using the squared difference between the *predicted vector* $H\overline{W}$ and the *observed vector* $\overline{y}$:

$$L = \frac{1}{2}||H\overline{W} - \overline{y}||^2$$

In order to reduce overfitting, one can add Tikhonov regularization to the objective function:

$$L = \frac{1}{2}||H\overline{W} - \overline{y}||^2 + \frac{\lambda}{2}||\overline{W}||^2 \tag{6.2}$$

Here, $\lambda > 0$ is the regularization parameter. By computing the partial derivative of L with respect to the elements of the weight vector, we obtain the following:

$$\frac{\partial L}{\partial \overline{W}} = H^T(H\overline{W} - \overline{y}) + \lambda \overline{W} = 0$$

The above derivative is written in matrix calculus notation where $\frac{\partial L}{\partial \overline{W}}$ refers to the following:

$$\frac{\partial L}{\partial \overline{W}} = \left[\frac{\partial L}{\partial w_1} \cdots \frac{\partial L}{\partial w_d}\right]^T \tag{6.3}$$

By re-adjusting the above condition, we obtain the following:

$$(H^T H + \lambda I)\overline{W} = H^T \overline{y}$$

When $\lambda > 0$, the matrix $H^T H + \lambda I$ is positive-definite and is therefore invertible. In other words, one obtains a simple solution for the weight vector in closed form:

$$\overline{W} = (H^T H + \lambda I)^{-1} H^T \overline{y} \tag{6.4}$$

Therefore, a simple matrix inversion is sufficient to find the weight vector, and backpropagation is completely unnecessary.

However, the reality is that the use of a closed-form solution is not viable in practice because the size of the matrix $H^T H$ is $m \times m$, which is large. In the particular case of RBF networks, the value of m is often larger than the input dimensionality of the data. This can make the matrix $H^T H$ extremely large. Therefore, one uses stochastic gradient descent to update the weight vector in practice. In such a case, the gradient-descent updates (with all training points) are as follows:

$$\begin{aligned}\overline{W} &\Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}} \\ &= \overline{W}(1 - \alpha\lambda) - \alpha H^T \underbrace{(H\overline{W} - \overline{y})}_{\text{Current Errors}}\end{aligned}$$

One can also choose to use mini-batch gradient descent in which the matrix H in the above update can be replaced with a random subset of rows H_r from H, corresponding to the mini-batch. This approach is equivalent to what would normally be used in a traditional neural network with mini-batch stochastic gradient descent. However, it is applied only to the weights of the connections incident on the output layer in this case.

Expression with Pseudo-Inverse

In the case in which the regularization parameter λ is set to 0, the weight vector $\overline{W}$ is defined as follows:

$$\overline{W} = (H^T H)^{-1} H^T \overline{y} \tag{6.5}$$

The matrix $(H^T H)^{-1} H^T$ is said to be the *pseudo-inverse* of the matrix H. The pseudo-inverse of the matrix H is denoted by H^+. Therefore, one can write the weight vector $\overline{W}^T$ as follows:

$$\overline{W} = H^+ \overline{y} \tag{6.6}$$

The pseudo-inverse is a generalization of the notion of an inverse for non-singular or rectangular matrices. In this particular case, $H^T H$ is assumed to be invertible, although the pseudo-inverse of H can be computed even in cases where $H^T H$ is not invertible. In the case where H is square and invertible, the pseudo-inverse is the same as its inverse.

6.2.3 Iterative Construction of Hidden Layer

This algorithm optimizes the selection of the prototypes for the hidden layer by using the accuracy of predictions. Therefore, the training process of the hidden layer is supervised, although the supervision is restricted to iterative selections from the original training points.

The algorithm chooses the prototype vector one by one from the training points in order to try them for accuracy, and select/discard them depending on the error of prediction.

The algorithm starts by building an RBF network with a single hidden node and trying each possible training point as a prototype in order to compute the prediction error. One then selects the prototype from the training points that minimizes the error of prediction. In the next iteration, one more prototype is added to the selected prototype in order to build an RBF network with two prototypes. As in the previous iteration, all $(n-1)$ remaining training points are tried as possible prototypes in order to add to the current bag of prototypes, and the criterion for adding to the bag is the minimization of prediction error. In the $(r+1)$th iteration, one tries all the $(n-r)$ remaining training points, and adds one of them to the bag of prototypes so that the prediction error is minimized. Some of the training points in the data are held out, and are not used in the computations of the predictions or as candidates for prototypes. These out-of-sample points are used in order to test the effect of adding a prototype to the error. In the early iterations, adding the best prototype to the hidden layer (based on prediction accuracy) lowers the overall accuracy from the previous iteration. At some point, the error on this held-out set can no longer be reduced by adding more prototypes to the hidden layer. An increase in error on the held-out test set is a sign of the fact that further increase in prototypes will increase overfitting. This is the point at which one terminates the algorithm.

The main problem with this approach is that it is extremely inefficient. In each iteration, one must run $O(n)$ training procedures, because one must try each training point in turn as a candidate for the hidden layer; this is computationally prohibitive for large training data sets. To improve efficiency, one can use the *orthogonal least-squares algorithm* [67]. This algorithm is similar to the one described above in the sense that the prototype vectors are added iteratively from the original training data set. However, the procedure with which the prototype is added is far more efficient. We refer the reader to [41] for more details.

6.2.4 Fully Supervised Learning of Hidden Layer

The iterative construction of the hidden layer represents a type of mild supervision in which the prototype vector is selected from one of the training points based on the effect to the overall prediction error. It is also possible to perform stronger types of supervision in which one can backpropagate in order to update the prototype vectors and the bandwidth. Consider the loss function L over the various training points:

$$L = \frac{1}{2}\sum_{i=1}^{n}(\overline{W} \cdot \overline{H}_i^T - y_i)^2 \tag{6.7}$$

Here, $\overline{H}_i$ represents the m-dimensional row vector of activations in the hidden layer for the ith training point $\overline{X}_i$, and $\overline{W}$ is the column vector of weights in the output layer.

The partial derivative with respect to each bandwidth σ_j can be computed as follows:

$$\begin{aligned}\frac{\partial L}{\partial \sigma_j} &= \sum_{i=1}^{n}(\overline{W} \cdot \overline{H}_i^T - y_i) w_j \frac{\partial \Phi_j(\overline{X}_i)}{\partial \sigma_j} \\ &= \sum_{i=1}^{n}(\overline{W} \cdot \overline{H}_i^T - y_i)\, w_j \Phi_j(\overline{X}_i)\, \frac{||\overline{X}_i - \overline{\mu}_j||^2}{\sigma_j^3}\end{aligned}$$

If all bandwidths σ_j are fixed to the same value σ, as is common in RBF networks, then the derivative can be computed using the same trick commonly used for handling shared

weights:

$$\frac{\partial L}{\partial \sigma} = \sum_{j=1}^{m} \frac{\partial L}{\partial \sigma_j} \cdot \underbrace{\frac{\partial \sigma_j}{\partial \sigma}}_{=1}$$
$$= \sum_{j=1}^{m} \frac{\partial L}{\partial \sigma_j}$$
$$= \sum_{j=1}^{m} \sum_{i=1}^{n} (\overline{W} \cdot \overline{H}_i^T - y_i)\, w_j \Phi_j(\overline{X}_i)\, \frac{||\overline{X}_i - \overline{\mu}_j||^2}{\sigma^3}$$

One can also compute a partial derivative with respect to each component of the prototype vector. Let μ_{jk} represent the kth element of $\overline{\mu}_j$. Similarly, let x_{ik} represent the kth component of the ith training vector $\overline{X}_i$. The partial derivative with respect to μ_{jk} is computed as follows:

$$\frac{\partial L}{\partial \mu_{jk}} = \sum_{i=1}^{n} (\overline{W} \cdot \overline{H}_i^T - y_i)\, w_j \Phi_j(\overline{X}_i)\, \frac{(x_{ik} - \mu_{jk})}{\sigma_j^2} \tag{6.8}$$

Using these partial derivatives, one can update the bandwidth and the prototype vectors together with the weights. Unfortunately, this type of strong approach to supervision does not seem to work very well. There are two main drawbacks with this approach:

1. An attractive characteristic of RBFs is that they are efficient to train, if unsupervised methods are used. However, this advantage is lost, if one resorts to full backpropagation. In general, the two-stage training of RBF is an efficiency feature of RBF networks.

2. The loss surface of RBFs has many local minima. This type of approach tends to get stuck in local minima from the point of view of generalization error.

Because of these characteristics of RBF networks, supervised training is rarely used. In fact, it has been shown in [353] that supervised training tends to increase the bandwidths and encourage generalized responses. When supervision is used, it should be used in a very controlled way by repeatedly testing performance on out-of-sample data in order to reduce the risk of overfitting.

6.3 Variations and Special Cases of RBF Networks

The above discussion only considers the case in which the supervised training is designed for numeric target variables. In practice, it is possible for the target variables to be binary. One possibility is to treat binary class labels in $\{-1, +1\}$ as numeric responses, and use the same approach of setting the weight vector according to Equation 6.4:

$$\overline{W} = (H^T H + \lambda I)^{-1} H^T \overline{y}$$

As discussed in section 3.2.2.1 of Chapter 3, this solution is also equivalent to the Fisher discriminant and the Widrow-Hoff method. The main difference is that these methods are being applied on a hidden layer of increased dimensionality, which promotes better results in more complex distributions. It is also helpful to examine other loss functions that are commonly used in feed-forward neural networks for classification.

6.3.1 Classification with Perceptron Criterion

Using the notations introduced in the previous section, the prediction of the ith training instance is given by $\overline{W} \cdot \overline{H}_i^T$. Here, $\overline{H}_i$ represents the m-dimensional row vector of activations in the hidden layer for the ith training instance $\overline{X}_i$, and $\overline{W}$ is the column vector of weights in the output layer. Then, as discussed in section 1.2.2 of Chapter 1, the perceptron criterion corresponds to the following loss function:

$$L = \max\{-y_i(\overline{W} \cdot \overline{H}_i^T), 0\} \tag{6.9}$$

In addition, a Tikhonov regularization term $\lambda||\overline{W}||^2$ with regularization parameter $\lambda > 0$ is often added to the loss function.

Then, for each mini-batch S of training instances, let S^+ represent the misclassified instances. The misclassified instances are defined as those for which the loss L is non-zero. For such instances, applying the sign function to $\overline{W} \cdot \overline{H}_i^T$ will yield a prediction with opposite sign to the observed label y_i.

Then, for each mini-batch S of training instances, the following updates are used for the misclassified instances in S^+:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{H}_i, y_i) \in S^+} y_i \overline{H}_i^T \tag{6.10}$$

Here, $\alpha > 0$ is the learning rate.

6.3.2 Classification with Hinge Loss

The hinge loss is used frequently in the support vector machine. Indeed, the use of hinge loss in the Gaussian RBF network can be viewed as a generalization of the support-vector machine. The hinge loss is a shifted version of the perceptron criterion:

$$L = \max\{1 - y_i(\overline{W} \cdot \overline{H}_i^T), 0\} \tag{6.11}$$

Because of the similarity in loss functions between the hinge loss and the perceptron criterion, the updates are also very similar. The main difference is that S^+ includes only misclassified points in the case of the perceptron criterion, whereas S^+ includes both misclassified points and marginally classified points in the case of hinge loss. This is because S^+ is defined by the set of points for which the loss function is non-zero, but (unlike the perceptron criterion) the hinge loss function is non-zero even for marginally classified points. Therefore, with this modified definition of S^+, the following updates are used:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{H}_i, y_i) \in S^+} y_i \overline{H}_i^T \tag{6.12}$$

Here, $\alpha > 0$ is the learning rate, and $\lambda > 0$ is the regularization parameter. Note that one can easily define similar updates for the logistic loss function (cf. Exercise 2).

6.3.3 Example of Linear Separability Promoted by RBF

The main goal of the hidden layer is to perform a transformation that promotes linear separability, so that even linear classifiers work well on the transformed data. Both the

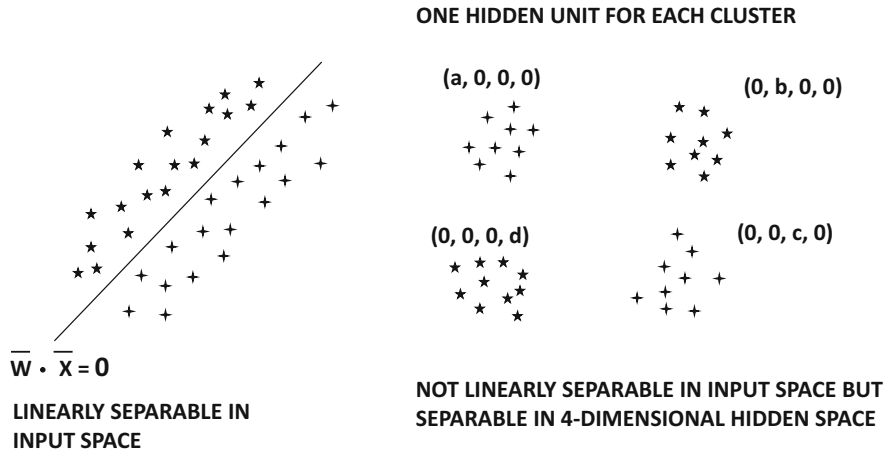


Figure 6.2: Revisiting Figure 1.4: The Gaussian RBF promotes separability because of the transformation to the hidden layer.

perceptron and the linear support vector machine with hinge loss are known to perform poorly when the classes are not linearly separable. The Gaussian RBF classifier is able to separate out classes that are not linearly separable in the input space when loss functions such as the perceptron criterion and hinge loss are used. The key to this separability is the local transformation created by the hidden layer. An important point is that a Gaussian kernel with a small bandwidth often results in a situation where only a small number of hidden units in particular local regions get activated to significant non-zero values, whereas the other values are almost zeros. This is because of the exponentially decaying nature of the Gaussian function, which takes on near-zero values outside a particular locality. The identification of prototypes with cluster centers often divides the space into local regions, in which significant non-zero activation is achieved only in small portions of the space. As a practical matter, each local region of the space is assigned its own feature, corresponding to the hidden unit that is activated most strongly by it.

Examples of two data sets are illustrated in Figure 6.2. These data sets were introduced in Chapter 1 to illustrate cases that the (traditional) perceptron can or cannot solve. The traditional perceptron of Chapter 1 is able to find a solution for the data set on the left, but does not work well for the data set on the right. However, the transformation used by the Gaussian RBF method is able to address this issue of separability for the clustered data set on the right. Consider a case in which each of the centroids of the four clusters in Figure 6.2 is used as a prototype. This will result in a 4-dimensional hidden representation of the data. Note that the hidden dimensionality is higher than the input dimensionality, which is common in these settings. With appropriate choice of bandwidth, only one hidden unit will be activated strongly corresponding to the cluster identifier to which the point belongs. The other hidden units will be activated quite weakly, and will be close to 0. This will result in a rather sparse representation, as shown in Figure 6.2. We have shown the approximate 4-dimensional representations for the points in each cluster. The values of a, b, c, and d in Figure 6.2 will vary over the different points in the corresponding cluster, although they will always be strongly non-zero compared to the other coordinates. Note that one of the

classes is defined by strongly non-zero values in the first and third dimensions, whereas the second class is defined by strongly non-zero values in the second and fourth dimensions. As a result, the weight vector $\overline{W} = [1, -1, 1, -1]$ will provide excellent non-linear separation between the two classes. The key point to understand is that the Gaussian RBF creates *local* features that result in separable distributions of the classes. This is exactly how a kernel support-vector machine achieves linear separability.

6.3.4 Application to Interpolation

One of the earliest applications of the Gaussian RBF was its use in interpolation of the value of a function over a set of points. The goal here is to perform *exact* interpolation of the provided points, so that the resulting function passes through all the input points. One can view interpolation as a special case of regression in which each training point is a prototype, and therefore the number of weights m in column vector $\overline{W}$ is exactly equal to the number of training examples n. In such cases, it is possible to find a n-dimensional weight vector $\overline{W}$ with zero error. In such a case, the activations $\overline{H}_1 \ldots \overline{H}_n$ represent n-dimensional row vectors. Therefore, the matrix H obtained by stacking these row vectors on top of each other has a size of $n \times n$. Let $\overline{y} = [y_1, y_2, \ldots y_n]^T$ be the n-dimensional column vector of observed variables.

In linear regression, one attempts to minimize the loss function $||H\overline{W} - \overline{y}||^2$ in order to determine $\overline{W}$. This is because the matrix H is not square, and the system of equations $H\overline{W} = \overline{y}$ is over-complete. However, in the case of linear interpolation, the matrix H is square, and the system of equations is no longer over-complete. Therefore, it is possible to find an exact solution (with zero loss) satisfying the following system of equations:

$$H\overline{W} = \overline{y} \tag{6.13}$$

It can be shown that this system of equations has a unique solution when the training points are distinct from one another [334]. The value of the weight vector $\overline{W}$ can then be computed as follows:

$$\overline{W} = H^{-1}\overline{y} \tag{6.14}$$

It is noteworthy that this equation is a special case of Equation 6.6 because the pseudo-inverse of a square and non-singular matrix is the same as its inverse. In the case where the matrix H is non-singular, one can simplify the pseudo-inverse as follows:

$$\begin{aligned} H^+ &= (H^T H)^{-1} H^T \\ &= H^{-1} \underbrace{(H^T)^{-1} H^T}_{I} \\ &= H^{-1} \end{aligned}$$

Therefore, the case of linear interpolation is a special case of least-squares regression. Stated in another way, least-squares regression is a form of noisy interpolation, where it is impossible to fit the function through all the training points because of the limited degrees of freedom in the hidden layer. Relaxing the size of the hidden layer to the training data size allows exact interpolation. Exact interpolation is not necessarily better for computing the function value of out-of-sample points, because it might be the result of overfitting.

6.4 Relationship with Kernel Methods

The RBF network gains its power by mapping the input points into a high-dimensional hidden space in which linear models are sufficient to model nonlinearities. This is the same principle used by kernel methods like kernel regression and kernel SVMs. In fact, it can be shown that certain special cases of the RBF network reduce to kernel regression and kernel SVMs.

6.4.1 Kernel Regression Is a Special Case of RBF Networks

The weight vector $\overline{W}$ in RBF networks is trained to minimize the squared loss of the following prediction function:

$$\hat{y}_i = \overline{W} \cdot \overline{H}_i^T = \sum_{j=1}^{m} w_j \Phi_j(\overline{X}_i) \tag{6.15}$$

Now consider the case in which the prototypes are the same as the training points, and therefore we set $\overline{\mu}_j = \overline{X}_j$ for each $j \in \{1 \ldots n\}$. Note that this approach is the same as that used in function interpolation, in which the prototypes are set to all the training points. Furthermore, each bandwidth σ is set to the same value. In such a case, one can write the above prediction function as follows:

$$\hat{y}_i = \sum_{j=1}^{n} w_j \exp\left(-\frac{||\overline{X}_i - \overline{X}_j||^2}{2\sigma^2}\right) \tag{6.16}$$

The exponentiated term on the right-hand side of Equation 6.16 can be written as the Gaussian kernel similarity between points $\overline{X}_i$ and $\overline{X}_j$. This similarity is denoted by $K(\overline{X}_i, \overline{X}_j)$. Therefore, the prediction function becomes the following:

$$\hat{y}_i = \sum_{j=1}^{n} w_j K(\overline{X}_i, \overline{X}_j) \tag{6.17}$$

This prediction function is exactly the same as that used in kernel regression with bandwidth σ, where the prediction function $\hat{y}_i^{kernel}$ is defined[1] in terms of the *Lagrange multipliers* λ_j instead of weight w_j (see, for example, [7]):

$$\hat{y}_i^{kernel} = \sum_{j=1}^{n} \lambda_j y_j K(\overline{X}_i, \overline{X}_j) \tag{6.18}$$

Furthermore, the (squared) loss function is the same in the two cases. Therefore, a one-to-one correspondence will exist between the Gaussian RBF solutions and the kernel regression solutions, so that setting $w_j = \lambda_j y_j$ leads to the same value of the loss function. Therefore, their optimal values will be the same as well. In other words, the Gaussian RBF network provides the same results as kernel regression in the special case where the prototype vectors are set to the training points. However, the RBF network is more powerful and general because it can choose different prototype vectors; therefore, the RBF network can model cases that are not possible with kernel regression. In this sense, it is helpful to view the RBF network as a flexible neural variant of kernel methods.

[1] A full explanation of the kernel regression prediction of Equation 6.18 is beyond the scope of this book. Readers are referred to [7].

6.4.2 Kernel SVM Is a Special Case of RBF Networks

Like kernel regression, the kernel support vector machine (SVM) is also a special case of RBF networks. As in the case of kernel regression, the prototype vectors are set to the training points, and the bandwidths of all hidden units are set to the same value of σ. Furthermore, the weights w_j are learned in order to minimize the hinge loss of the prediction.

In such a case, it can be shown that the prediction function of the RBF network is as follows:

$$\hat{y}_i = \text{sign}\left\{\sum_{j=1}^{n} w_j \exp\left(-\frac{||\overline{X}_i - \overline{X}_j||^2}{2\sigma^2}\right)\right\} \tag{6.19}$$

$$\hat{y}_i = \text{sign}\left\{\sum_{j=1}^{n} w_j K(\overline{X}_i, \overline{X}_j)\right\} \tag{6.20}$$

It is instructive to compare this prediction function with that used in kernel SVMs (see, for example, [7]) with the Lagrange multipliers λ_j:

$$\hat{y}_i^{kernel} = \text{sign}\left\{\sum_{j=1}^{n} \lambda_j y_j K(\overline{X}_i, \overline{X}_j)\right\} \tag{6.21}$$

This prediction function is of a similar form as that used in kernel SVMs, with the exception of a slight difference in the variables used. The hinge-loss is used as the objective function in both cases. By setting $w_j = \lambda_j y_j$ one obtains the same result in both cases in terms of the value of the loss function. Therefore, the optimal solutions in the kernel SVM and the RBF network will also be related according to the condition $w_j = \lambda_j y_j$. In other words, the kernel SVM is also a special case of RBF networks. Note that the weight w_j can also be considered the coefficient of each data point, when the *representer theorem* is used in kernel methods [7].

Observations

One can extend the arguments above to other linear models, such as the kernel Fisher discriminant and kernel logistic regression, by changing the loss function. In fact, the kernel Fisher discriminant can be obtained by simply using the binary variables as the targets and then applying kernel regression technique. However, since the Fisher discriminant works under the assumption of centered data, a bias needs to be added to the output layer to absorb any offsets from uncentered data. Therefore, the RBF network can simulate virtually any kernel method by choosing an appropriate loss function. A key point is that the RBF network provides more flexibility than kernel regression or classification. For example, one has much more flexibility in choosing the number of nodes in the hidden layer, as well as the number of prototypes. Choosing the prototypes wisely in a more economical way helps in both accuracy and efficiency. There are a number of key trade-offs associated with these choices. Increasing the number of hidden units increases the complexity of the modeled function. It can be useful for modeling difficult functions, but it can also cause overfitting. One way of choosing the number of hidden units is to hold out a portion of the data, and estimate the accuracy of the model on the held-out set with different numbers of hidden units. The number of hidden units is then set to a value that optimizes this accuracy.

6.5 Summary

This chapter introduces radial basis function (RBF) networks, which represent a fundamentally different way of using the neural network architecture. Unlike feed-forward networks, the hidden layer and output layer are trained in a somewhat different way. The training of the hidden layer is unsupervised, whereas that of the output layer is supervised. The hidden layer usually has a larger number of nodes than the input layer. The key idea is to transform the data points into high-dimensional space with the use of locality-sensitive transformations, so that the transformed points become linearly separable. The approach can be used for classification, regression, and linear interpolation by changing the nature of the loss function. In classification, one can use different types of loss functions such as the Widrow-Hoff loss, the hinge loss, and the logistic loss. Special cases of different loss functions specialize to well-known kernel methods such as kernel SVMs and kernel regression. The RBF network has rarely been used in recent years, and it has become a forgotten category of neural architectures. However, it has significant potential to be used in any scenario where kernel methods are used. Furthermore, it is possible to combine this approach with feed-forward architectures by using multi-layered representations following the first hidden layer.

6.6 Bibliographic Notes and Software Resources

RBF networks were proposed by Broomhead and Lowe [51] in the context of function interpolation. The separability of high-dimensional transformations is shown in Cover's work [87]. A review of RBF networks may be found in [374]. The books by Bishop [41] and Haykin [192] also provide good treatments of the topic. An overview of radial basis functions is provided in [57]. The proof of universal function approximation with RBF networks is provided in [183, 378]. An analysis of the approximation properties of RBF networks is provided in [379].

Efficient training algorithms for RBF networks are described in [358, 440]. An algorithm for learning the center locations in RBF networks is proposed in [549]. The use of decision trees to initialize RBF networks is discussed in [264]. The orthogonal least-squares algorithm was proposed in [67]. Early comparisons of supervised and unsupervised training of RBF networks are provided in [353]. According to this analysis, full supervision seems to increase the likelihood of the network getting trapped in local minima. Some ideas on improving the generalization power of RBF networks are provided in [42]. Incremental RBF networks are discussed in [130]. A detailed discussion of the relationship between RBF networks and kernel methods is provided in [449].

6.7 Exercises

Some exercises require additional knowledge about machine learning that is not discussed in this book. Exercises 5, 7, and 8 require additional knowledge of kernel methods, spectral clustering, and outlier detection.

1. Consider the following variant of radial basis function networks in which the hidden units take on either 0 or 1 values. The hidden unit takes on the value of 1, if the distance to a prototype vector is less than σ. Otherwise it takes on the value of 0.

Discuss the relationship of this method to RBF networks, and its relative advantages/disadvantages.

2. Suppose that you use the sigmoid activation in the final layer to predict a binary class label as a probability in the output node of an RBF network. Set up a negative log-likelihood loss for this setting. Derive the gradient-descent updates for the weights in the final layer. How does this approach relate to the logistic regression methods discussed in Chapter 3? In which case will this approach perform better than logistic regression?

3. Discuss why an RBF network is a supervised variant of a nearest-neighbor classifier.

4. Discuss how you can extend the three multi-class models discussed in Chapter 2 to RBF networks. In particular discuss the extension of the (a) multi-class perceptron, (b) Weston-Watkins SVM, and (c) softmax classifier with RBF networks. Discuss how these models are more powerful than the ones discussed in Chapter 2.

5. Propose a method to extend RBF networks to unsupervised learning with autoencoders. What will you reconstruct in the output layer? A special case of your approach should be able to roughly simulate kernel singular value decomposition.

6. Suppose that you change your RBF network so that you keep only the top-k activations in the hidden layer, and set the remaining activations to 0. Discuss why such an approach will provide improved classification accuracy with limited data.

7. Combine the top-k method of constructing the RBF layer in Exercise 6 with the RBF autoencoder in Exercise 5 for unsupervised learning. Discuss why this approach will create representations that are better suited to clustering. Discuss the relationship of this method with spectral clustering.

8. The manifold view of outliers is to define them as points that do not naturally fit into the nonlinear manifolds of the training data. Discuss how you can use RBF networks for unsupervised outlier detection.

9. Suppose that instead of using the RBF function in the hidden layer, you use dot products between prototypes and data points for activation. Show that a special case of this setting reduces to a linear perceptron.

10. Discuss how you can modify the RBF autoencoder in Exercise 5 to perform semi-supervised classification, when you have a lot of unlabeled data, and a limited amount of labeled data.

11. **RBF backpropagation:** It can sometimes be useful to backpropagate through the RBF layer to compute the sensitivity $\partial o / \partial x_j$ of class output probability o to each input feature value x_j. Suppose that the ith hidden feature h_i for the RBF layer is computed as $h_i = \exp(-\|\overline{X} - \overline{\mu}_i\|^2 / 2\sigma^2)$. Derive an expression for $\partial h_i / \partial x_j$.

12. **RBF network as attention mechanism:** Suppose each input to an RBF network is normalized to have an L_2-norm of 1, and the RBF prototypes are sampled from the data. Assume that the RBF layer uses the standard RBF function shown in Exercise 11. Discuss the similarities and differences of such a layer with the softmax layer that takes as input each $\overline{X} \cdot \overline{\mu}_i$ for all prototypes i. How can you add some simple (unparameterized) computational units after the RBF layer and select σ appropriately to obtain softmax outputs? In other words, RBF networks *attend* to fixed prototypes.

Chapter 7

Restricted Boltzmann Machines

"Available energy is the main object at stake in the struggle for existence and the evolution of the world." – Ludwig Boltzmann

7.1 Introduction

The restricted Boltzmann machine (RBM) is a fundamentally different model from the feed-forward network. Conventional neural networks are input-output mapping networks where a set of inputs is mapped to a set of outputs. On the other hand, RBMs are networks in which the probabilistic states of a network are learned for a set of inputs, which is useful for *unsupervised* modeling. While a feed-forward network minimizes a loss function of a prediction (computed from *observed* inputs) with respect to an *observed* output, a restricted Boltzmann machine models the joint probability distribution of the observed attributes together with some hidden attributes. Whereas traditional feed-forward networks have *directed* edges corresponding to the flow of computation from input to output, RBMs are *undirected* networks because they are designed to learn probabilistic *relationships* rather than input-output mappings. Restricted Boltzmann machines are probabilistic models that create latent representations of the underlying data points. Although an autoencoder can also be used to construct latent representations, the representations are deterministic; a Boltzmann machine creates *stochastic* hidden representations. As a result, the RBM requires a fundamentally different way of training and using it. At their core, RBMs are unsupervised models that generate latent feature representations of the data points, although there are many ways of extending them to the supervised case. It is noteworthy that RBMs usually work with binary states in their most natural form, although it is possible to work with other data types. Most of the discussion in this chapter will be restricted to units with binary states.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_7

Historical Perspective

Restricted Boltzmann machines have evolved from a classical model in the neural networks literature, which is referred to as the *Hopfield network*. This network contains nodes containing binary states, which represent binary attribute values in the training data. The Hopfield network creates a *deterministic* model of the relationships among the different attributes by using weighted edges between nodes. Eventually, the Hopfield network evolved into the notion of a Boltzmann machine, which uses *probabilistic* states to represent the Bernoulli distributions of the binary attributes. The Boltzmann machine contains both visible states and hidden states. The visible states model the distributions of the observed data points, whereas the hidden states model the distribution of the latent (hidden) variables. The parameters of the connections among the various states regulate their joint distribution. The goal is to learn the model parameters so that the likelihood of the model is maximized. The Boltzmann machine is a member of the family of (undirected) probabilistic graphical models. Eventually, the Boltzmann machine evolved into the *restricted* Boltzmann Machine (RBM). The main difference between the Boltzmann machine and the restricted Boltzmann machine is that the latter only allows connections between hidden units and visible units. This simplification is very useful from a practical point of view, because it allows the design of more efficient training algorithms. The RBM is a special case of the class of probabilistic graphical models known as *Markov random fields*.

In the initial years, RBMs were considered too slow to train and were therefore not very popular. However, at the turn of the century, faster algorithms were proposed for this class of models. Furthermore, they received some prominence as one of the ensemble components of the entry [431] winning the Netflix prize contest [602]. The successful training of deep networks with RBMs preceded successful training experiences with conventional neural networks; similar ideas were then generalized to conventional networks.

Chapter Organization

This chapter is organized as follows. The next section will introduce Hopfield networks, which was the precursor to the Boltzmann family of models. The Boltzmann machine is introduced in section 7.3. Restricted Boltzmann machines are introduced in section 7.4. Applications of restricted Boltzmann machines are discussed in section 7.5. The use of RBMs for generalized data types beyond binary representations is discussed in section 7.6. The process of stacking multiple restricted Boltzmann machines in order to create deep networks is discussed in section 7.7. A summary is given in section 7.8.

7.2 Hopfield Networks

Hopfield networks were proposed in 1982 [217] as a model to store memory. A Hopfield network is an undirected network, in which the d units (or neurons) are indexed by values drawn from $\{1 \ldots d\}$. Each connection is of the form (i, j), where each i and j is a neuron drawn from $\{1 \ldots d\}$. Each connection (i, j) is undirected, and is associated with a weight $w_{ij} = w_{ji}$. Although all pairs of nodes are assumed to have connections between them, setting w_{ij} to 0 has the effect of dropping the connection (i, j). The weight w_{ii} is set to 0, and therefore there are no self-loops. Each neuron i is associated with state s_i. An important assumption in the Hopfield network is that each s_i is a binary value drawn from $\{0, 1\}$, although one can use other conventions such as $\{-1, +1\}$. The ith node also has a bias b_i associated with it; large values of b_i encourage the ith state to be 1. The Hopfield

network is an undirected model of symmetric relationships between attributes, and therefore the weights always satisfy $w_{ij} = w_{ji}$.

Each binary state in the Hopfield network corresponds to a dimension in the (binary) training data set. Therefore, if a d-dimensional training data set needs to be memorized, we need a Hopfield network with d units. The ith state in the network corresponds to the ith bit in a particular training example. The values of the states represent the binary attribute values from a training example. The weights in the Hopfield network are its parameters; large positive weights between pairs of states are indicative of high degree of positive correlation in state values, whereas large negative weights are indicative of high negative correlation. An example of a Hopfield network with an associated training data set is shown in Figure 7.1. In this case, the Hopfield network is fully connected, and the six visible states correspond to the six binary attributes in the training data.

The Hopfield network uses an optimization model to learn the weight parameters so that the weights can capture that positive and negative relationships among the attributes of the training data set. The objective function of a Hopfield network is also referred to as its *energy function*, which is analogous to the loss function of a traditional feed-forward neural network. The energy function of a Hopfield network is set up in such a way that minimizing this function encourages node pairs connected with large positive weights to have the same state, and pairs connected with large negative weights to have different states. The training phase of a Hopfield network, therefore, learns the weights of edges in order to minimize the energy when the states in the Hopfield network are fixed to the binary attribute values in the individual training points. Therefore, learning the weights of the Hopfield network implicitly builds an unsupervised model of the training data set. The energy E of a particular combination of states $\overline{s} = (s_1, \ldots s_d)$ of the Hopfield network can be defined as follows:

$$E = -\sum_i b_i s_i - \sum_{i,j:i<j} w_{ij} s_i s_j \tag{7.1}$$

The term $-b_i s_i$ encourages $s_i = 1$ when b_i is positive in order to minimize energy. Similarly, the term $-w_{ij} s_i s_j$ encourages $s_i = s_j$ when $w_{ij} > 0$. In other words, positive weights will cause state "attraction" and negative weights will cause state "repulsion." For a small training data set, this type of modeling results in memorization, which enables one to retrieve training data points from similar, incomplete, or corrupted query points by exploring local minima of the energy function near these query points. In other words, by learning the weights of a Hopfield network, one is implicitly memorizing the training examples, although there is a relatively conservative limit of the number of examples that can be memorized from a Hopfield network containing d units. This limit is also referred to as the *capacity* of the model.

Optimal State Configurations of a Trained Network

A trained Hopfield network contains many local optima, each of which corresponds to either a memorized point from the training data, or a representative point in a dense region of the training data. Before discussing the training of the weights of the Hopfield network, we will discuss the methodology for finding the local energy minimum of a Hopfield network when the trained weights are already given. A local minimum is defined as a combination of states in which flipping any particular bit of the network does not reduce the energy further. The training process sets the weights in such a way that the instances in the training data tend to be local minima in the Hopfield network.

Finding the optimal state configuration helps the Hopfield network in recalling memories. The Hopfield network inherently learns *associative memories* because, given an input set of states (i.e., input pattern of bits), it repeatedly flips bits to improve the objective function until it finds a pattern where one cannot improve the objective function further. This local minimum (final combination of states) is often only a few bits away from the starting pattern (initial set of states), and therefore one *recalls* a closely related pattern at which a local minimum is found. Furthermore, this final pattern is often a member of the training data set (because the weights were learned using that data). In a sense, Hopfield networks provide a route towards *content-addressable memory*.

Given a starting combination of states, how can one learn the closest local minimum once the weights have already been fixed? One can use a threshold update rule to update each state in the network in order to move it towards the global energy minimum. In order to understand this point, let us compare the energy of the network between the cases when the state s_i is set to 1, and the one in which s_i is set to 0. Therefore, one can substitute two different values of s_i into Equation 7.1 to obtain the following value of the *energy gap*:

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j \tag{7.2}$$

This value must be larger than 0 in order for a flip of state s_i from 0 to 1 to be attractive. Therefore, one obtains the following update rule for each state s_i:

$$s_i = \begin{cases} 1 & \text{if } \sum_{j:j \neq i} w_{ij} s_j + b_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

The above rule is iteratively used to test each state s_i and then flip the state if needed to satisfy the condition. If one is given the weights and the biases, repeatedly using the state updates reaches a local energy minimum.

The local minima of a Hopfield network depend on its trained weights. Therefore, in order to "recall" a memory, one only has to provide a d-dimensional vector similar to the stored memory, and the Hopfield network will find the local minimum that is similar to this point by using it as a starting state. This type of associative memory recall is also common in humans, who often retrieve memories through a similar process of association. One can also provide a partial vector of initial states and use it to recover other states. Consider the Hopfield network shown in Figure 7.1. Note that the weights are set in such a way

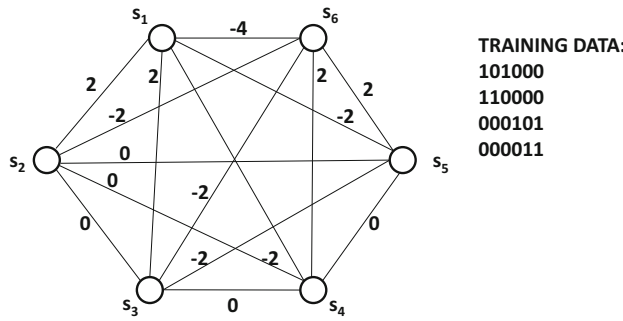


Figure 7.1: A Hopfield network with 6 visible states corresponding to 6-dimensional training data.

that each of the four training vectors in the figure will have low energy. However, it is not guaranteed that the local minima will always correspond to the points in the training data. However, the local minima do correspond to some key characteristics of the training data. For example, consider the minimum corresponding to 111000. It is noteworthy that the first three bits are positively correlated, whereas the last three bits are also positively correlated. As a result, this minimum value of 111000 does reflect a broad pattern in the underlying data even though it is not explicitly present in the training data. This is because the weights of this network are closely related to the patterns in the training data. For example, the elements within the first three bits and last three bits are each positively correlated within their respective groups. Furthermore, there are negative correlations *across* the two sets of elements. Consequently, the edges *within* each of the sets $\{s_1, s_2, s_3\}$ and $\{s_4, s_5, s_6\}$ tend to be positive, and those *across* these two sets are negative. Setting the weights in this data-specific way is the task of the training phase (cf. section 7.2.1).

The iterative state update rule will arrive at one of the many local minima of the Hopfield network, depending on the initial state vector. Each of these local minima might be one of the learned *associative memories* from the training data set, and the closest memory to the initial state vector will be reached. These memories are implicitly stored in the weights learned during the training phase. The Hopfield network is similar to how humans recall memories in terms of *associating* their past seen patterns approximately with current patterns. However, it is possible for the Hopfield network to arrive at a closely related pattern, when it results from the merging of related training patterns into a single (deeper) minimum. For example, if the training data contains 1110111101 and 1110111110, the Hopfield network might learn 1110111111 as a local minimum. Therefore, in some queries, one might recover a pattern that is a small number of bits away from a pattern actually present in the training data. However, this is only a form of model generalization in which the Hopfield network is storing representative "cluster" centers instead of individual training points. The model starts generalizing instead of memorizing when the amount of data exceeds the capacity of the model; after all, Hopfield networks build unsupervised models from data.

The Hopfield network can be used for recalling associative memories, correcting corrupted data, or for attribute completion. For recalling associative memories or cleaning corrupted data, one uses the (possibly corrupted) target vector for recall as the starting state. The final state is the recalled output (or cleaned output). In attribute completion, the state vector is initialized by setting observed states to their known values and unobserved states randomly. At this point, only the unobserved states are updated to convergence. The bit values of these states at convergence provide the completed representation.

7.2.1 Training a Hopfield Network

For a given training data set containing n instances, one needs to learn the weights, so that the local minima of this network lie near instances (or dense regions) of the training data set. Hopfield networks are trained with the *Hebbian learning rule*. According to the biological motivation of Hebbian learning, a synapse between two neurons is strengthened when the neurons on either side of the synapse have highly correlated outputs. Let $x_{ij} \in \{0, 1\}$ represent the jth bit of the ith training point. The weights of the network are set using the Hebbian learning rule:

$$w_{ij} = 4 \sum_{k=1}^{n} (x_{ki} - 0.5) \cdot (x_{kj} - 0.5) \tag{7.4}$$

One way of understanding this rule is that if two bits, i and j, in the training data are positively correlated, then the value $(x_{ki} - 0.5) \cdot (x_{kj} - 0.5)$ will usually be positive. As a result, the weights between the corresponding units will also be set to positive values. On the other hand, if two bits generally disagree, then the weights will be set to negative values. The bias b_i can be set by assuming that it is a weight between two units, such that one of them is always on, and therefore, it is equal to $2\sum_k (x_{kj} - 0.5)$.

One can also update w_{ij} incrementally with only the kth training data point as follows:

$$w_{ij} \Leftarrow w_{ij} + 4(x_{ki} - 0.5) \cdot (x_{kj} - 0.5) \quad \forall i, j$$

The bias b_i can be updated by assuming that it represents the weight between an always-on state and the ith state:

$$b_i \Leftarrow b_i + 2(x_{ki} - 0.5) \quad \forall i$$

In cases where the convention is to draw the state vectors from $\{-1, +1\}$, the above rule simplifies to the following:

$$w_{ij} \Leftarrow w_{ij} + x_{ki}x_{kj} \quad \forall i, j$$
$$b_i \Leftarrow b_i + x_{ki} \quad \forall i$$

There are other learning rules, such as the *Storkey learning rule*, that are commonly used. Refer to the bibliographic notes.

Capacity of a Hopfield Network

What is the size of the training data that a Hopfield network with d visible units can store without causing errors in associative recall? It can be shown that the *storage capacity* of a Hopfield network with d units is only about $0.15 \cdot d$ training examples. Since each training example contains d bits, it follows that the Hopfield network can store only about $0.15\, d^2$ bits. This is not an efficient form of storage because the number of weights in the network is given by $d(d-1)/2 = O(d^2)$. Furthermore, the weights are not binary and they can be shown to require $O(\log(d))$ bits. When the number of training examples is large, many errors will be made (in associative recall). These errors represent the *generalized* predictions from more data. Although it might seem that this type of generalization is useful for machine learning, there are limitations in using Hopfield networks for such applications.

7.2.2 Building a Toy Recommender and Its Limitations

Hopfield networks are often used for memorization-centric applications rather than the typical machine-learning applications requiring generalization. In order to understand the limits of a Hopfield network, we will consider the problem of binary collaborative filtering with *implicit feedback* data in which each user is associated d binary attributes corresponding to whether or not they have watched each of d possible movies. Consider a situation in which the user Bob has watched movies *Shrek* and *Aladdin*, whereas the user Alice has watched *Gandhi*, *Nero*, and *Terminator*. It is easy to construct a fully connected Hopfield network on the universe of all movies and set the watched states to 1 and all other states to 0. This configuration can be used for each training point in order to update the weights. Of course, this approach can be extremely expensive if the base number of states (movies) is very large. For a database containing 10^6 movies, we would have 10^{12} edges, most of which will connect states containing zero values. This is because such type of implicit feedback data is often sparse, and most states will take on zero values.

One way of addressing this problem is to use *negative sampling*. In this approach, each user has their own Hopfield network containing their watched movies and a small sample of the movies that were not watched by them. For example, one might randomly sample 20 unwatched movies (of Alice) and create a Hopfield network containing $20 + 3 = 23$ states (including the watched movies). Bob's Hopfield network will containing $20 + 2 = 22$ states, and the unwatched samples might also be quite different. However, for pairs of movies that are common between the two networks, the weights will be shared. During training, all edge weights are initialized to 0. One can use repeated iterations of training over the different Hopfield networks to learn their shared weights (with the same algorithm discussed earlier). The main difference is that iterating over the different training points will lead to iterating over different Hopfield networks, each of which contains a small subset of the base network. Typically, only a small subset of the 10^{12} edges will be present in each of these networks, and most edges will never be encountered in any network. Such edges will implicitly be assumed to have weights of zero.

Now imagine a user Mary, who has watched *E.T.* and *Shrek*. We would like to recommend movies to this user. We use the full Hopfield network with only the non-zero edges present. We initialize the states for *E.T.* and *Shrek* to 1, and all other states to 0. Subsequently, we allow the updates of all states (other than *E.T.* and *Shrek*) in order to identify the minimum energy configuration of the Hopfield network. All states that are set to 1 during the updates can be recommended to the user. However, we would ideally like to have an *ordering* of the top recommended movies. One way of providing an ordering of all movies is to use the *energy gap* between the two states of each movie in order to rank the movies. The energy gap is computed only after the minimum energy configuration has been found. This approach is, however, quite naive because the final configuration of the Hopfield network is a deterministic one containing binary values, whereas the extrapolated values can only be estimated in terms of *probabilities*. For example, it would be much more natural to use some function of the energy gap (e.g., sigmoid) in order to create probabilistic estimations. Furthermore, it would be helpful to be able to capture correlated sets of movies with some notion of latent (or hidden) states. Clearly, we need techniques in order to increase the expressive power of the Hopfield network.

7.2.3 Increasing the Expressive Power of the Hopfield Network

Although it is not standard practice, one can add *hidden units* to a Hopfield network to increase its expressive power. The hidden states serve the purpose of capturing the latent structure of the data. The weights of connections between hidden and visible units will capture the relationship between the latent structure and the training data. In some cases, it is possible to approximately represent the data only in terms of a small number of hidden states. For example, if the data contains two tightly knit clusters, one can capture this setting in two hidden states. Consider the case in which we enhance the Hopfield network of Figure 7.1 and add two hidden units. The resulting network is shown in Figure 7.2. The edges with near-zero weights have been dropped from the figure for clarity. Even though the original data is defined in terms of six bits, the two hidden units provide a *hidden* representation of the data in terms of two bits. This hidden representation is a compressed version of the data, which tells us something about the pattern at hand. In essence, all patterns are compressed to the pattern 10 or 01, depending on whether the first three bits or the last three bits dominate the training pattern. If one fixes the hidden states of the Hopfield network to 10 and randomly initializes the visible states, then one would often obtain the pattern 111000 on repeatedly using the state-wise update rule of Equation 7.3. One also

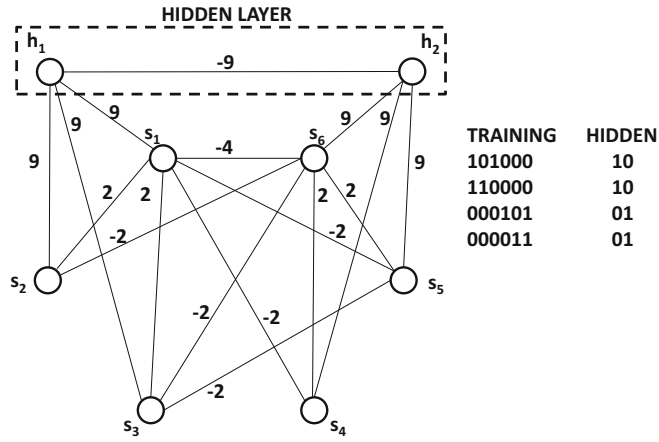


Figure 7.2: The Hopfield network with two hidden nodes

obtains the pattern 0001111 as the final resting point when one starts with the hidden state 01. Notably, the patterns 000111 and 111000 are close approximations of the two types of patterns in the data, which is what one would expect from a compression technique. If we provide an incomplete version of the visible units, and then iteratively update the other states with the update rule of Equation 7.3, one would often arrive at either 000111 and 111000 depending on how the bits in the incomplete representation are distributed. If we add hidden units to a Hopfield network *and* allow the states to be probabilistic (rather than deterministic), we obtain a Boltzmann machine. This is the reason that Boltzmann machines can be viewed as *stochastic Hopfield networks with hidden units*.

7.3 The Boltzmann Machine

Throughout this section, we assume that the Boltzmann machine contains a total of $q = (m+d)$ states, where d is the number of visible states and m is the number of hidden states. A particular state configuration is defined by the value of the state vector $\overline{s} = (s_1 \ldots s_q)$. If one explicitly wants to demarcate the visible and hidden states in $\overline{s}$, then the state vector $\overline{s}$ can be written as the pair $(\overline{v}, \overline{h})$, where $\overline{v}$ denotes the set of visible units and $\overline{h}$ denotes the set of hidden units. The states in $(\overline{v}, \overline{h})$ represent exactly the same set as $\overline{s} = \{s_1 \ldots s_q\}$, except that the visible and hidden units are explicitly demarcated in the former.

The Boltzmann machine is a probabilistic generalization of a Hopfield network. A Hopfield network *deterministically* sets each state s_i to either 1 or 0, depending on whether the energy gap ΔE_i of the state s_i is positive or negative. Recall that the energy gap of the ith unit is defined as the difference in energy between its two configurations (with other states being fixed to pre-defined values):

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j \tag{7.5}$$

The Hopfield network deterministically sets the value of s_i to 1, when the energy gap is positive. On the other hand, a Boltzmann machine assigns a *probability* to s_i depending on

the energy gap. Positive energy gaps are assigned probabilities that are larger than 0.5. The probability of state s_i is defined by applying the sigmoid function to the energy gap:

$$P(s_i = 1|s_1, \ldots s_{i-1}, s_{i+1}, s_q) = \frac{1}{1 + \exp(-\Delta E_i)} \tag{7.6}$$

Note that the state s_i is now a Bernoulli random variable and a zero energy gap leads to a probability of 0.5 for each binary outcome of the state.

For a particular set of parameters w_{ij} and b_i, the Boltzmann machine defines a probability distribution over various state configurations. The energy of a particular configuration $\overline{s} = (\overline{v}, \overline{h})$ is denoted by $E(\overline{s}) = E([\overline{v}, \overline{h}])$, and is defined in a similar way to the Hopfield network as follows:

$$E(\overline{s}) = -\sum_i b_i s_i - \sum_{i,j:i<j} w_{ij} s_i s_j \tag{7.7}$$

However, these configurations are only probabilistically known in the case of the Boltzmann machine (according to Equation 7.6). The conditional distribution of Equation 7.6 follows from the definition of the unconditional probability $P(\overline{s})$ of configuration $\overline{s}$:

$$P(\overline{s}) \propto \exp(-E(\overline{s})) = \frac{1}{Z}\exp(-E(\overline{s})) \tag{7.8}$$

The normalization factor Z ensures that probabilities over all possible configurations sum to 1:

$$Z = \sum_{\overline{s}} \exp(-E(\overline{s})) \tag{7.9}$$

The normalization factor Z is also referred to as the *partition function*. In general, the explicit computation of the partition function is hard, because it contains an exponential number of terms corresponding to all possible configurations of states. Because of the intractability of the partition function, exact computation of $P(\overline{s}) = P(\overline{v}, \overline{h})$ is not possible. Nevertheless, the computation of many types of conditional probabilities (e.g., $P(\overline{v}|\overline{h})$) is possible, because such conditional probabilities are ratios and the intractable normalization factor gets canceled out from the computation. For example, the conditional probability of Equation 7.6 follows from the more fundamental definition of the probability of a configuration (cf. Equation 7.8) as follows:

$$\begin{aligned} P(s_i = 1|s_1, \ldots s_{i-1}, s_{i+1}, s_q) &= \frac{P(s_1, \ldots s_{i-1}, \overbrace{1}^{s_i}, s_{i+1}, s_q)}{P(s_1, \ldots s_{i-1}, \underbrace{1}_{s_i}, s_{i+1}, s_q) + P(s_1, \ldots s_{i-1}, \underbrace{0}_{s_i}, s_{i+1}, s_q)} \\ &= \frac{\exp(-E_{s_i=1})}{\exp(-E_{s_i=1}) + \exp(-E_{s_i=0})} = \frac{1}{1 + \exp(E_{s_i=1} - E_{s_i=0})} \\ &= \frac{1}{1 + \exp(-\Delta E_i)} = \text{Sigmoid}(\Delta E_i) \end{aligned}$$

This is the same condition as Equation 7.8. Since states are probabilistic, we can sample from their joint probability distributions to create new data points that look like the original data. Therefore. the Boltzmann machine can ne used as a generative model. Many generative models in machine learning (e.g., Gaussian mixture models for clustering) use a sequential process of first sampling the hidden state(s) from a prior, and then generating

visible observations conditionally on the hidden state(s). This is not the case in the Boltzmann machine, in which the dependence between all pairs of states is *undirected*; the visible states depend as much on the hidden states as the hidden states depend on visible states. As a result, the generation of data with a Boltzmann machine can be more challenging than in many other generative models.

7.3.1 How a Boltzmann Machine Generates Data

The main challenge in data generation from a Boltzmann machine is caused by the circular dependencies among states (based on Equation 7.6). Therefore, a probabilistic sampling process is used by the Boltzmann machine (based on Equation 7.6). A Boltzmann machine iteratively samples the states using a conditional distribution generated from the state values in the previous iteration until *thermal equilibrium* is reached. The notion of thermal equilibrium means that the observed frequencies of sampling various attribute values represent their long-term steady-state probability distributions. The process of reaching thermal equilibrium works as follows. We start at a random set of states, use Equation 7.6 to compute their conditional probabilities, and then sample the values of the states again using these probabilities. Note that we can iteratively generate s_i by using $P(s_i = 1|s_1 \ldots s_{i-1}, s_{i+1}, \ldots s_q)$ in Equation 7.6. After running this process for a long time, the sampled values of the visible states provide us with random samples of generated data points. The time required to reach thermal equilibrium is referred to as the *burn-in time* of the procedure. This approach is referred to as *Gibbs sampling* or *Markov Chain Monte Carlo (MCMC) sampling*.

At thermal equilibrium, the generated points will represent the model captured by the Boltzmann machine. Note that the dimensions in the generated data points will be correlated with one another depending on the weights between various states. States with large weights between them will tend to be heavily correlated. For example, in a text-mining application in which the states correspond to the presence of words, there will be correlations among words belonging to a topic. Therefore, if a Boltzmann machine has been trained properly on a text data set, it will generate vectors containing these types of word correlations at thermal equilibrium, even when the states are randomly initialized. It is noticeable that even generating a set of data points with the Boltzmann machine is a more complicated process compared to many other probabilistic models. For example, generating data points from a Gaussian mixture model only requires to sample points directly from the probability distribution of a sampled mixture component. On the other hand, the undirected nature of the Boltzmann machine forces us to run the process to thermal equilibrium just to generate samples. It is, therefore, an even more difficult to task to learn the weights between states for a given training data set.

7.3.2 Learning the Weights of a Boltzmann Machine

In a Boltzmann machine, we want to learn the weights in such a way so as to maximize the log-likelihood of the specific training data set at hand. The log-likelihoods of individual states are computed by using the logarithm of the probabilities in Equation 7.8. Therefore, by taking the logarithm of Equation 7.8, we obtain the following:

$$\log[P(\overline{s})] = -E(\overline{s}) - \log(Z) \tag{7.10}$$

Therefore, computing $\frac{\partial \log[P(\overline{s})]}{\partial w_{ij}}$ requires the computation of the negative derivative of the energy, although we have an additional term involving the partition function. The energy

function of Equation 7.7 is linear in the weight w_{ij} with coefficient of $-s_i s_j$. Therefore, the partial derivative of the energy with respect to the weight w_{ij} is $-s_i s_j$. As a result, one can show the following:

$$\frac{\partial \log[P(\overline{s})]}{\partial w_{ij}} = \langle s_i, s_j \rangle_{data} - \langle s_i, s_j \rangle_{model} \tag{7.11}$$

Here, $\langle s_i, s_j \rangle_{data}$ represents the averaged value of $s_i s_j$ obtained by running the generative process of section 7.3.1, when the visible states are clamped to attribute values in a training point. The averaging is done over a mini-batch of training points. Similarly, $\langle s_i, s_j \rangle_{model}$ represents the averaged value of $s_i s_j$ at thermal equilibrium without fixing visible states to training points and simply running the generative process of section 7.3.1. In this case, the averaging is done over multiple instances of running the process to thermal equilibrium. Intuitively, we want to strengthen the weights of edges between states that tend to be turned on together when the visible states are fixed to the training data points. This is precisely what is achieved by the update above, which uses the data- and model-centric difference in the value of $\langle s_i, s_j \rangle$. From the above discussion, it is clear that two types of samples need to be generated in order to perform the updates:

1. **Data-centric samples:** The first type of sample fixes the visible states to a randomly chosen vector from the training data set. The hidden states are initialized to random values drawn from Bernoulli distribution with probability 0.5. Then the probability of each hidden state is recomputed according to Equation 7.6. Samples of the hidden states are regenerated from these probabilities. This process is repeated for a while, so that thermal equilibrium is reached. The values of the hidden variables at this point provide the required samples. Note that the visible states are clamped to the corresponding attributes of the relevant training data vector, and therefore they do not need to be sampled.

2. **Model-centric samples:** The second type of sample does not put any constraints on fixing states to training data points, and one simply wants samples from the unrestricted model. The approach is the same as discussed above, except that both the visible and hidden states are initialized to random values, and updates are continuously performed until thermal equilibrium is reached.

These samples help us create an update rule for the weights. From the first type of sample, one can compute $\langle s_i, s_j \rangle_{data}$, which represents the correlations between the states of nodes i and j, when the visible vectors are fixed to a vector in the training data $\mathcal{D}$ and the hidden states are allowed to vary. Since a mini-batch of training vectors is used, one obtains multiple samples of the state vectors. The value of $\langle s_i, s_j \rangle$ is computed as the average product over all such state vectors that are obtained from Gibbs sampling. Similarly, one can estimate the value of $\langle s_i, s_j \rangle_{model}$ using the average product of s_i and s_j from the model-centric samples obtained from Gibbs sampling. Once these values have been computed, the following update is used:

$$w_{ij} \Leftarrow w_{ij} + \alpha \underbrace{(\langle s_i, s_j \rangle_{data} - \langle s_i, s_j \rangle_{model})}_{\text{Partial derivative of log probability}} \tag{7.12}$$

The update rule for the bias is similar, except that the state s_j is set to 1. One can achieve this by using a dummy bias unit that is visible and is connected to all states:

$$b_i \Leftarrow b_i + \alpha (\langle s_i, 1 \rangle_{data} - \langle s_i, 1 \rangle_{model}) \tag{7.13}$$

Note that the value of $\langle s_i, 1 \rangle$ is simply the average of the sampled values of s_i for a mini-batch of training examples from either the data-centric samples or the model-centric samples.

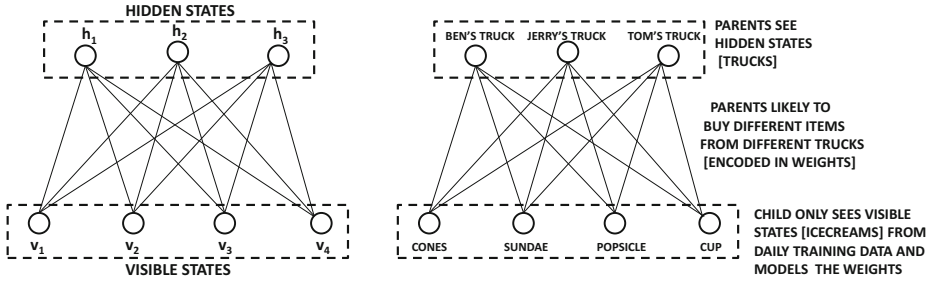
This approach is similar to the Hebbian update rule of a Hopfield net, except that we are also removing the effect of model-centric correlations in the update. The removal of model-centric correlations is required to account for the effect of the partition function within the expression of the log probability in Equation 7.10. The main problem with the aforementioned update rule is that it is slow in practice. This is because of the Monte Carlo sampling procedure, which requires a large number of samples to reach thermal equilibrium.

7.4 Restricted Boltzmann Machines

In the Boltzmann machine, the connections among hidden and visible units can be arbitrary. For example, two hidden states might contain edges between them, and so might two visible states. This type of generalized assumption creates unnecessary complexity. A natural special case of the Boltzmann machine is the *restricted* Boltzmann machine (RBM), which is bipartite, and the connections are allowed only between hidden and visible units. An example of a restricted Boltzmann machine is shown in Figure 7.3(a). In this particular example, there are three hidden nodes and four visible nodes. Each hidden state is connected to one or more visible states, although there are no connections between pairs of hidden states, and between pairs of visible states. The restricted Boltzmann machine is also referred to as a *harmonium* [477].

We assume that the hidden units are $h_1 \ldots h_m$ and the visible units are $v_1 \ldots v_d$. The bias associated with the visible node v_i be denoted by $b_i^{(v)}$, and the bias associated with hidden node h_j is denoted by $b_j^{(h)}$. Note the superscripts in order to distinguish between the biases of visible and hidden nodes. The weight of the edge between visible node v_i and hidden node h_j is denoted by w_{ij}. The notations for the weights are also slightly different for the restricted Boltzmann machine (compared to the Boltzmann machine) because the hidden and visible units are indexed separately. For example, we no longer have $w_{ij} = w_{ji}$ because the first index i always belongs to a visible node and the second index j belongs to a hidden node. It is important to keep these notational differences in mind while extrapolating the equations from the previous section.

In order to provide better interpretability, we will use a running example throughout this section, which we refer to as the example of "Alice's ice-cream trucks" based on the Boltzmann machine in Figure 7.3(b). Imagine a situation in which the training data corre-



(a) A Boltzmann machine (b) Alice's ice-cream trucks

Figure 7.3: A Restricted Boltzmann machine. Note the *restriction* of there being no interactions among either visible or hidden units.

sponds to four bits representing the ice-creams received by Alice from her parents each day. These represent the visible states in our example. Therefore, Alice can collect 4-dimensional training points, as she receives (between 0 and 4) ice-creams of different types each day. However, the ice-creams are bought for Alice by her parents from one[1] or more of three trucks shown as the hidden states in the same figure. The identity of these trucks is hidden from Alice, although she knows that there are three trucks from which her parents procure the ice-creams (and more than one truck can be used to construct a single day's ice-cream set). Alice's parents are indecisive people, and their decision-making process is unusual because they change their mind about the selected ice-creams after selecting the trucks and vice versa. The likelihood of a particular ice-cream being picked depends on the trucks selected as well as the weights to these trucks. Similarly, the likelihood of a truck being selected depends on the ice-creams that one intends to buy and the same weights. Therefore, Alice's parents can keep changing their mind about selecting ice-creams after selecting trucks and about selecting trucks after selecting ice-creams (for a while) until they reach a final decision each day. As we will see, this *circular* relationship is the characteristic of undirected models, and process used by Alice's parents is similar to Gibb's sampling.

The use of the bipartite restriction greatly simplifies inference algorithms in RBMs, while retaining much of the application-centric power. If we know all the values of the visible units (as is common when a training data point is provided), the probabilities of the hidden units can be computed in one step without having to go through the arduous task of Gibbs sampling. The probability of each hidden unit taking on the value of 1 can be expressed directly as a logistic function of the values of visible units. In other words, we can apply Equation 7.6 to the RBM to obtain the following:

$$P(h_j = 1|\overline{v}) = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{7.14}$$

This result follows directly from Equation 7.6, which relates the state probabilities to the energy gap ΔE_j between $h_j = 0$ and $h_j = 1$. The value of ΔE_j is $b_j + \sum_i v_i w_{ij}$ when the visible states are observed. The main difference from an unrestricted Boltzmann machine is that the right-hand side of the above equation does not contain any (unknown) hidden variables and it only contains visible variables. This relationship is also useful in creating a reduced representation of each training vector, once the weights have been learned. Specifically, for a Boltzmann machine with m hidden units, one can set the value of the jth hidden value to the probability computed in Equation 7.14. Note that such an approach provides a real-valued reduced representation of the binary data. One can also write the above equation using a sigmoid function:

$$P(h_j = 1|\overline{v}) = \text{Sigmoid}\left(b_j^{(h)} + \sum_{i=1}^{d} v_i w_{ij}\right) \tag{7.15}$$

One can also use a sample of the hidden states to generate the data points in one step. This is because the relationship between the visible units and the hidden units is similar in the undirected and bipartite architecture of the RBM. In other words, we can use Equation 7.6 to obtain the following:

$$P(v_i = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{7.16}$$

[1]This example is tricky in terms of semantic interpretability for the case in which no trucks are selected. Even in that case, the probabilities of various ice-creams turn out to be non-zero depending on the bias. One can explain such cases by adding a dummy truck that is always selected.

One can also express this probability in terms of the sigmoid function:

$$P(v_i = 1|\overline{h}) = \text{Sigmoid}\left(b_i^{(v)} + \sum_{j=1}^{m} h_j w_{ij}\right) \tag{7.17}$$

One nice consequence of using the sigmoid is that it is often possible to create a closely related feed-forward network with sigmoid activation units in which the weights learned by the Boltzmann machine are leveraged in a directed computation with input-output mappings. The weights of this network are then fine-tuned with backpropagation. We will give examples of this approach in the application section.

Note that the weights encode the affinities between the visible and hidden states. A large positive weight implies that the two states are likely to be on together. For example, in Figure 7.3(b), it might be possible that the parents are more likely to buy cones and sundae from Ben's truck, whereas they are more likely to buy popsicles and cups from Tom's truck. These propensities are encoded in the weights, which regulate both visible state selection and hidden state selection in a circular way. The *circular* nature of the relationship creates challenges, because the relationship between ice-cream choice and truck choice runs both ways; it is the raison d'etre for Gibb's sampling. Although Alice might not know which trucks the ice-creams are coming from, she will notice the resulting correlations among the bits in the training data. In fact, if the weights of the RBM are known by Alice, she can use Gibb's sampling to generate 4-bit points representing "typical" examples of ice-creams she will receive on future days. Even the weights of the model can be learned by Alice from examples, which is the essence of an unsupervised generative model. Given the fact that there are 3 hidden states (trucks) and enough examples of 4-dimensional training data points, Alice can learn the relevant weights and biases between the visible ice-creams and hidden trucks. An algorithm for doing this is discussed in the next section.

7.4.1 Training the RBM

Computation of the weights of the RBM is achieved using a similar type of learning rule as that used for Boltzmann machines. In particular, it is possible to create an efficient algorithm based on mini-batches. The weights w_{ij} are initialized to small values. For the current set of weights w_{ij}, they are updated as follows:

- *Positive phase:* The algorithm uses a mini-batch of training instances, and computes the probability of the state of each hidden unit in exactly one step using Equation 7.14. Then a single sample of the state of each hidden unit is generated from this probability. This process is repeated for each element in a mini-batch of training instances. The correlation between these different training instances of v_i and generated instances of h_j is computed; it is denoted by $\langle v_i, h_j \rangle_{pos}$. This correlation is essentially the average product between each such pair of visible and hidden units.

- *Negative phase:* In the negative phase, the algorithm starts with randomly initialized states and uses Equations 7.14 and 7.16 repeatedly to thermal equilibrium to compute the probabilities of the visible and hidden units. These probabilities are used to draw samples if v_i and h_j, and the entire process is repeated multiple times. The multiple samples are used to compute the average product $\langle v_i, h_j \rangle_{neg}$ in the same way as the positive phase.

- One can then use the same type of update as is used in Boltzmann machines:

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg} \right)$$
$$b_i^{(v)} \Leftarrow b_i^{(v)} + \alpha \left(\langle v_i, 1 \rangle_{pos} - \langle v_i, 1 \rangle_{neg} \right)$$
$$b_j^{(h)} \Leftarrow b_j^{(h)} + \alpha \left(\langle 1, h_j \rangle_{pos} - \langle 1, h_j \rangle_{neg} \right)$$

  Here, $\alpha > 0$ denotes the learning rate. Each $\langle v_i, h_j \rangle$ is estimated by averaging the product of v_i and h_j over the mini-batch, although the values of v_i and h_j are computed in different ways in the positive and negative phases, respectively. Furthermore, $\langle v_i, 1 \rangle$ represents the average value of v_i in the mini-batch, and $\langle 1, h_j \rangle$ represents the average value of h_j in the mini-batch.

It is helpful to interpret the updates above in terms of Alice's trucks in Figure 7.3(b). When the weights of certain visible bits (e.g., cones and sundae) are highly correlated, the above updates will tend to push the weights in directions that these correlations can be explained by the weights between the trucks and the ice-creams. For example, if the cones and sundae are highly correlated but all other correlations are very weak, it can be explained by high weights between each of these two types of ice-creams and a single truck. In practice, the correlations will be far more complex, as will the patterns of the underlying weights.

7.4.2 Contrastive Divergence Algorithm

One issue with the above approach is the time required to reach thermal equilibrium and generate negative samples. However, it turns out that it is possible to run the Monte Carlo sampling for only a short time *starting by fixing the visible states to a training data point from the mini-batch* and still obtain a good approximation of the gradient. The fastest variant of the contrastive divergence approach uses a *single* additional iteration of Monte Carlo sampling (over what is done in the positive phase) in order to generate the samples of the hidden and visible states. First, the hidden states are generated by fixing the visible units to a training point (which is already accomplished in the positive phase), and then the visible units are generated again (exactly once) from these hidden states using Monte Carlo sampling. The values of the visible units are used as the sampled negative states in lieu of the ones obtained at thermal equilibrium. The hidden units are generated again using these visible units. Thus, the main difference between the positive and negative phase is only of the number of iterations that one runs the approach starting with the same initialization of visible states to training points. In the positive phase, we use only half an iteration of simply computing the hidden states. In the negative phase, we use at least one *additional* iteration (so that visible states are recomputed from hidden states and hidden states generated again). This difference in the number of iterations is what causes the contrastive divergence between the state distributions in the two cases. The intuition is that an increased number of iterations causes the distribution to move away (i.e., diverge) from the data-conditioned states to what is proposed by the current weight vector. Therefore, the value of $(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg})$ in the update quantifies the amount of contrastive divergence. This fastest variant of the contrastive divergence algorithm is referred to as CD_1 because it uses a single (additional) iteration in order to generate the negative samples. Of course, using such an approach is only an approximation to the true gradient. One can improve the accuracy of contrastive divergence by increasing the number of additional iterations to k, in which the data is reconstructed k times. This approach is referred to as CD_k. Increased values of k lead to better gradients at the expense of speed.

In the early iterations, using CD_1 is good enough, although it might not be helpful in later phases. Therefore, a natural approach is to progressively increase the value of k, while applying CD_k in training. One can summarize this process as follows:

1. In the early phase of gradient-descent, the weights are very inexact. In each iteration, only one additional step of contrastive divergence is sufficient, because only a rough direction of descent is needed to improve the inexact weights. Therefore, even if CD_1 is executed, one will be able to obtain a good direction in most cases.

2. As the gradient descent nears a better solution, higher accuracy is needed. Therefore, two or three steps of contrastive divergence are used (i.e., CD_2 or CD_3). In general, one can double the number of Markov chain steps after a fixed number of gradient descent steps. Another approach advocated in [485] is to create the value of k in CD_k by 1 after every 10,000 steps. The maximum value of k used in [485] was 20.

An excellent practical guide for training restricted Boltzmann machines may be found in [203]. This guide discusses several practical issues such as initialization, tuning, and updates. The following provides a brief overview of some of these practical issues.

Practical Issues and Improvisations

Although we have always assumed that the Monte Carlo sampling procedure generates discrete samples, some iterations of Monte Carlo sampling improvise in order to directly use *computed* probabilities (cf. Equations 7.14 and 7.16) instead of discrete samples. This reduces the noise in training, because probability values retain more information than binary samples. Specifically, the following improvisations are used:

- *Improvisations in sampling hidden states:* The final iteration of CD_k computes hidden states as probability values according to Equation 7.14 for positive and negative samples. Therefore, the value of h_j used for computing $\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg}$ would always be a real value for both positive and negative samples. This real value is a fraction because of the use of the sigmoid function in Equation 7.14.

- *Improvisations in sampling visible states:* Therefore, the improvisations for Monte Carlo sampling of visible states are always associated with the computation of $\langle v_i, h_j \rangle_{neg}$ rather than $\langle v_i, h_j \rangle_{pos}$ because visible states are always fixed to the training data. For the negative samples, the Monte Carlo procedure *always* computes probability values of visible states according to Equation 7.16 over *all* iterations rather than using 0-1 values. This is not the case for the hidden states, which are always binary until the very last iteration.

Using probability values iteratively rather than sampled binary values introduces bias, and does not reach correct thermal equilibrium. However, the contrastive divergence algorithm is an approximation anyway, and this type of approach reduces noise (variance) at the expense of some bias.

The weights can be initialized from a Gaussian distribution with zero mean and a standard deviation of 0.01. Large values of initial weights can speed up learning, but might lead to a slightly worse model in the end. The visible biases are initialized to $\log(p_i/(1-p_i))$, where p_i is the fraction of data points in which the ith dimension takes on the value of 1. The values of the hidden biases are initialized to 0. The mini-batch size should be somewhere between 10 and 100. The order of the examples should be randomized. For supervised settings with class labels, the mini-batch should be selected in a stratified way, so that the proportions of labels in the batch roughly match the whole data.

7.5 Applications of Restricted Boltzmann Machines

In this section, we will study several applications of restricted Boltzmann machines. These methods have been very successful for a variety of unsupervised applications, although they are also used for supervised applications. When using an RBM in a real-world application, a mapping from input to output is often required, whereas a vanilla RBM is only designed to learn probability distributions. The input-to-output mapping is often achieved by constructing a feed-forward network with weights derived from the learned RBM. In other words, one can often derive a traditional neural network that is *associated* with the original RBM.

Here, we will like to discuss the differences between the notions of the *state* of a node in the RBM, and the *activation* of that node in the associated neural network. The state of a node is a binary value sampled from the Bernoulli probabilities defined by Equations 7.14 and 7.16. On the other hand, the activation of a node in the associated neural network is the probability value derived from the use of the sigmoid function in Equations 7.14 and 7.16. Many applications use the activations in the nodes of the associated neural network, rather than the states in the original RBM after the training. Note that the final step in the contrastive divergence algorithm also leverages the activations of the nodes rather than the states while updating the weights. In practical settings, the activations are more information-rich and are therefore useful. The use of activations is consistent with traditional neural network architectures, in which backpropagation can be used. The use of a final phase of backpropagation is crucial in being able to apply the approach to supervised applications. In most cases, the critical role of the RBM is to perform unsupervised feature learning. Therefore, the role of the RBM is often only one of pretraining in the case of supervised learning. In fact, pretraining is one of the important historical contributions of the RBM.

7.5.1 Dimensionality Reduction and Data Reconstruction

The most basic function of the RBM is that of dimensionality reduction and unsupervised feature engineering. The hidden units of an RBM contain a reduced representation of the data. However, we have not yet discussed how one can reconstruct the original representation of the data with the use of an RBM (much like an autoencoder). In order to understand the reconstruction process, we first need to understand the equivalence of the undirected RBM with directed graphical models [259], in which the computation occurs in a particular direction. Materializing a directed probabilistic graph is the first step towards materializing a traditional neural network (derived from the RBM) in which the discrete probabilistic sampling from the sigmoid can be replaced with real-valued sigmoid activations.

Although an RBM is an undirected graphical model, one can "unfold" an RBM in order to create a directed model in which the inference occurs in a particular direction. In general, an undirected RBM can be shown to be equivalent to a directed graphical model with an infinite number of layers. The unfolding is particularly useful when the visible units are fixed to specific values because the number of layers in the unfolding collapses to exactly twice the number of layers in the original RBM. Furthermore, by replacing the discrete probabilistic sampling with continuous sigmoid units, this directed model functions as a virtual autoencoder, which has both an encoder portion and a decoder portion. Although the weights of an RBM have been trained using discrete probabilistic sampling, they can also be used in this related neural network with some fine tuning. This is a heuristic approach to convert what has been learned from a Boltzmann machine (i.e., the weights) into the

initialized weights of a traditional neural network with sigmoid units.

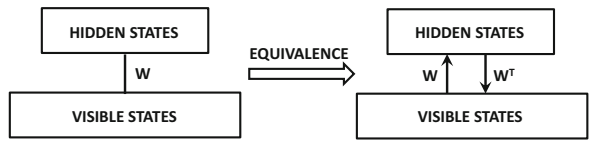
An RBM can be viewed as an undirected graphical model that uses the same weight matrix to learn $\overline{h}$ from $\overline{v}$ as it does from $\overline{v}$ to $\overline{h}$. If one carefully examines Equations 7.14 and 7.16, one can see that they are very similar. The main difference is that these equations uses different biases, and they use the transposes of each other's weight matrices. In other words, one can rewrite Equations 7.14 and 7.16 in the following form for some function $f(\cdot)$:

$$\overline{h} \sim f(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v} \sim f(\overline{h}, \overline{b}^{(v)}, W^T)$$

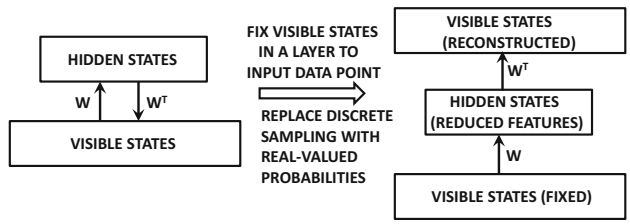
The function $f(\cdot)$ is typically defined by the sigmoid function in binary RBMs, which constitute the predominant variant of this class of models. Ignoring the biases, one can replace the undirected graph of the RBM with two directed links, as shown in Figure 7.4(a). Note that the weight matrices in the two directions are W and W^T, respectively. However, if we fix the visible states to the training points, we can perform just two iterations of these operations to reconstruct the visible states with *real-valued* approximations. In other words, we approximate this trained RBM with a traditional neural network by replacing discrete sampling with continuous-valued sigmoid activations (as a heuristic). This conversion is shown in Figure 7.4(b). In other words, instead of using the sampling operation of "$\sim$," we replace the samples with the probability values:

$$\overline{h} = f(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v}' = f(\overline{h}, \overline{b}^{(v)}, W^T)$$

Note that $\overline{v}'$ is the reconstructed version of $\overline{v}$ and it will contain real values (unlike the binary states in $\overline{v}$). In this case, we are working with real-valued activations rather than discrete samples. Because sampling is no longer used and all computations are performed in terms of expectations, we need to perform only one iteration of Equation 7.14 in order to learn the reduced representation. Furthermore, only one iteration of Equation 7.16 is required to learn the reconstructed data. The prediction phase works only in a single direction from the



(a) Equivalence of directed and undirected relationships



(b) Discrete graphical model to approximate real-valued neural network

Figure 7.4: Using trained RBM to approximate trained autoencoder

input point to the reconstructed data, and is shown on the right-hand side of Figure 7.4(b). We modify Equations 7.14 and 7.16 to define the states of this traditional neural network as real values:

$$\hat{h}_j = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{7.18}$$

For a setting with a total of $m \ll d$ hidden states, the real-valued reduced representation is given by $(\hat{h}_1 \ldots \hat{h}_m)$. This first step of creating the hidden states is equivalent to the encoder portion of an autoencoder, and these values are the expected values of the binary states. One can then apply Equation 7.16 to these *probabilistic values* (without creating Monte-Carlo instantiations) in order to reconstruct the visible states as follows:

$$\hat{v}_i = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_j \hat{h}_j w_{ij})} \tag{7.19}$$

Although $\hat{h}_j$ does represent the expected value of the jth hidden unit, applying the sigmoid function again to this real-valued version of $\hat{h}_j$ only provides a rough approximation to the expected value of v_i. Nevertheless, the real-valued prediction $\hat{v}_i$ is an approximate reconstruction of v_i. Note that in order to perform this reconstruction we have used similar operations as traditional neural networks with sigmoid units rather than the troublesome discrete samples of probabilistic graphical models. Therefore, we can now use this related neural network as a good starting point for fine-tuning the weights with traditional backpropagation. This type of reconstruction is similar to the reconstruction used in the autoencoder architecture discussed in Chapter 3.

On first impression, it makes little sense to train an RBM when similar goals can be achieved with a traditional autoencoder. However, this broad approach of deriving a traditional neural network with a trained RBM is particularly useful when working with stacked RBMs (cf. section 7.7). The training of a stacked RBM does not face the same challenges as those associated with deep neural networks, especially the ones related with the vanishing and exploding gradient problems. Just as the simple RBM provides an excellent initialization point for the shallow autoencoder, the stacked RBM also provides an excellent starting point for a deep autoencoder [208]. This principle led to the development of the idea of pretraining with RBMs before conventional pretraining methods were developed without the use of RBMs. As discussed in this section, one can also use RBMs for other reduction-centric applications such as collaborative filtering and topic modeling.

7.5.2 RBMs for Collaborative Filtering

The previous section shows how restricted Boltzmann machines are used as alternatives to the autoencoder for unsupervised modeling and dimensionality reduction. However, as discussed in section 3.5 of Chapter 3, dimensionality reduction methods are also used for a variety of related applications like collaborative filtering. In the following, we will provide an RBM-centric alternative to the recommendation technique described in section 3.5 of Chapter 3. This approach is based on the technique proposed in [431], and it was one of the ensemble components of the winning entry in the Netflix prize contest.

One of the challenges in working with ratings matrices is that they are incompletely specified. This tends to make the design of a neural architecture for collaborative filtering more difficult than traditional dimensionality reduction. Recall from the discussion in section 3.5 that modeling such incomplete matrices with a traditional neural network also faces the same challenge. In that section, it was shown how one could create a different training

instance *and* a different neural network for each user, depending on which ratings are observed by that user. All these different neural networks share weights. An exactly similar approach is used with the restricted Boltzmann machine, in which one training case and one RBM is defined for each user. However, in the case of the RBM, one additional problem is that the units are binary, whereas ratings can take on values from 1 to 5. Therefore, we need some way of working with the additional constraint.

In order to address this issue, the hidden units in the RBM are allowed to be 5-way softmax units in order to correspond to rating values from 1 to 5. In other words, the hidden units are defined in the form of a one-hot encoding of the rating. One-hot encodings are naturally modeled with softmax, which defines the probabilities of each possible position. The ith softmax unit corresponds to the ith movie and the probability of a particular rating being given to that movie is defined by the distribution of softmax probabilities. Therefore, if there are d movies, we have a total of d such one-hot encoded ratings. The values of the corresponding binary values of the one-hot encoded visible units are denoted by $v_i^{(1)}, \ldots v_i^{(5)}$. Note that only one of the values of $v_i^{(k)}$ can be 1 over fixed i and varying k. The hidden layer is assumed to contain m units. The weight matrix has a separate parameter for each of the multinomial outcomes of the softmax unit. Therefore, the weight between visible unit i and hidden unit j for the outcome k is denoted by $w_{ij}^{(k)}$. In addition, we have 5 biases for the visible unit i, which are denoted by $b_i^{(k)}$ for $k \in \{1, \ldots, 5\}$. The hidden units only have a single bias, and the bias of the jth hidden unit is denoted by b_j (without a superscript). The architecture of the RBM for collaborative filtering is illustrated in Figure 7.5. This example contains $d = 5$ movies and $m = 2$ hidden units. In this case, the RBM architectures of two users, Sayani and Bob, are shown in the figure. In the case of Sayani, she has specified ratings for only two movies. Therefore, a total of $2 \times 2 \times 5 = 20$ connections will be present in her case, even though we have shown only a subset of them to avoid clutter in the figure. In the case of Bob, he has four observed ratings, and therefore his network will contain a total of $4 \times 2 \times 5 = 40$ connections. Note that both Sayani and Bob have rated the movie *E.T.*, and therefore the connections from this movie to the hidden units will share weights between the corresponding RBMs. The probabilities of the binary hidden states are defined with the use of the sigmoid function:

$$P(h_j = 1 | \overline{v}^{(1)} \ldots \overline{v}^{(5)}) = \frac{1}{1 + \exp(-b_j - \sum_{i,k} v_i^{(k)} w_{ij}^k)} \tag{7.20}$$

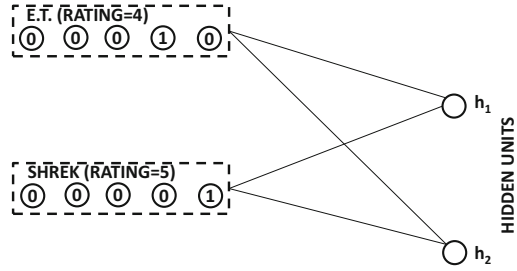
The main difference from Equation 7.14 is that the visible units also contain a superscript to correspond to the different rating outcomes. Otherwise, the condition is virtually identical. The probabilities of the visible units are defined even more differently from the traditional RBM model with the softmax function:

$$P(v_i^{(k)} = 1 | \overline{h}) = \frac{\exp(b_i^{(k)} + \sum_j h_j w_{ij}^{(k)})}{\sum_{r=1}^{5} \exp(b_i^{(r)} + \sum_j h_j w_{ij}^{(r)})} \tag{7.21}$$

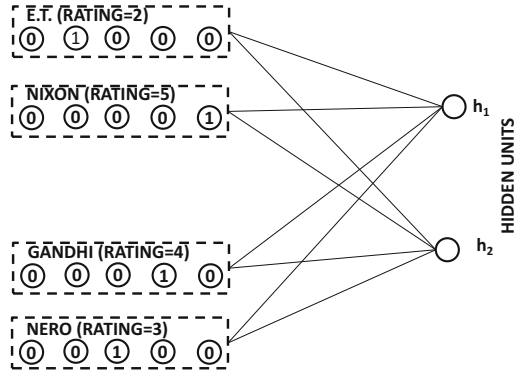
The training is done with Monte Carlo sampling. The main difference from the previously discussed methods is that the visible states are generated from a multinomial model. The corresponding updates for training the weights are as follows:

$$w_{ij}^{(k)} \Leftarrow w_{ij}^{(k)} + \alpha \left(\langle v_i^{(k)}, h_j \rangle_{pos} - \langle v_i^{(k)}, h_j \rangle_{neg} \right) \quad \forall k \tag{7.22}$$

Note that only the weights of the *observed* visible units to all hidden units are updated for a single training example (i.e., user). In other words, the Boltzmann machine that is used is



(a) RBM architecture for user Sayani (Observed Ratings: *E.T.* and *Shrek*)



(b) RBM architecture for user Bob (Observed Ratings: *E.T.*, *Nixon*, *Gandhi*, and *Nero*)

Figure 7.5: The RBM architectures of two users are shown based on their observed ratings. It is instructive to compare this figure with the conventional neural architecture shown in Figure 3.13 in Chapter 3. In both cases, weights are shared by user-specific networks.

different for each user in the data, although the weights are shared across the different users. Examples of the Boltzmann machines for two different training examples are illustrated in Figure 7.5, and the architectures for Bob and Sayani are different. However, the weights for the units representing *E.T.* are shared. This type of approach is also used in the traditional neural architecture of section 3.5 in which the neural network used for each training example is different. As discussed in that section, the traditional neural architecture is equivalent to a matrix factorization technique. The Boltzmann machine tends to give somewhat different ratings predictions from matrix factorization techniques, although the accuracy is similar.

Making Predictions

Once the weights have been learned, they can be used for making predictions. However, the predictive phase works with real-valued activations rather than binary states, much like a traditional neural network with sigmoid and softmax units. First, one can use Equation 7.20 in order to learn the probabilities of the hidden units. Let the probability that the jth hidden unit is 1 be denoted by $\hat{p}_j$. Then, the probabilities of *unobserved* visible units are computed using Equation 7.21. The main problem in computing Equation 7.21 is that it is defined in terms of the values of the hidden units, which are only known in the form of probabilities according to Equation 7.20. However, one can simply replace each h_j with $\hat{p}_j$

in Equation 7.21 in order to compute the probabilities of the visible units. Note that these predictions provide the probabilities of each possible rating value of each item. These probabilities can also be used to compute the expected value of the rating if needed. Although this approach is approximate from a theoretical point of view, it works well in practice and is extremely fast. By using these real-valued computations, one is effectively converting the RBM into a traditional neural network architecture with logistic units for hidden layers and softmax units for the input and output layers. Although the original paper [431] does not mention it, it is even possible to tune the weights of this network with backpropagation (cf. Exercise 1).

The RBM approach works as well as the traditional matrix factorization approach, although it tends to give different types of predictions. This type of diversity is an advantage from the perspective of using an ensemble-centric approach. Therefore, the results can be combined with the matrix factorization approach in order to yield the improvements that are naturally associated with an ensemble method. Ensemble methods generally show better improvements when diverse methods of similar accuracy are combined. A number of tricks such as *conditional factoring* [431] can be used in order to improve the accuracy of the RBM.

7.5.3 Using RBMs for Classification

The most common way to use RBMs for classification is as a pretraining procedure. In other words, a Boltzmann machine is first used to perform unsupervised feature engineering. The RBM is then unrolled into a related encoder-decoder architecture according to the approach described in section 7.5.1. This is a traditional neural network with sigmoid units, whose weights are derived from the unsupervised RBM rather than backpropagation. The encoder portion of this neural network is topped with an output layer for class prediction. The weights of this neural network are then fine-tuned with backpropagation. Such an approach can even be used with *stacked RBMs* (cf. section 7.7) to yield a deep classifier. This methodology of initializing a (conventional) deep neural network with an RBM was one of the first approaches for pretraining deep networks.

There is, however, another alternative approach to perform classification with the RBM, which integrates RBM training and inference more tightly with the classification process. This approach is somewhat similar to the collaborative filtering methodology discussed in the previous section. The collaborative-filtering problem is also referred to as *matrix completion* because the missing entries of an incompletely specified matrix are predicted. The use of RBMs for recommender systems provides some useful hints about their use in classification. This is because classification can be viewed as a simplified version of the matrix completion problem in which we create a single matrix out of both the training and test rows, and the missing values belong to a particular column of the matrix. This column corresponds to the class variable. Furthermore, all the missing values are present in the test rows in the case of classification, whereas the missing values could be present anywhere in the matrix in the case of recommender systems. In classification, all features are observed for the rows corresponding to training points, which simplifies the modeling (compared to collaborative filtering in which a complete set of features is typically not observed for any row).

We assume that the input data contains d binary features. The class label has k discrete values, which corresponds to the multiway classification problem. The classification problem can be modeled by the RBM by defining the hidden and visible features as follows:

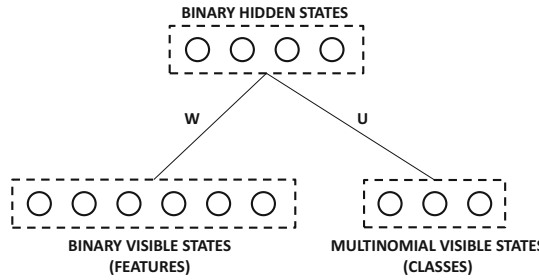


Figure 7.6: The RBM architecture for classification

1. The visible layer contains two types of nodes corresponding to the features and the class label, respectively. There are d binary units corresponding to features, and there are k binary units corresponding to the class label. However, only one of these k binary units can take on the value of 1, which corresponds to a one-hot encoding of the class labels. This encoding of the class label is similar to the approach used for encoding the ratings in the collaborative-filtering application. The visible units for the features are denoted by $v_1^{(f)} \ldots v_d^{(f)}$, whereas the visible units for the class labels are denoted by $v_1^{(c)} \ldots v_k^{(c)}$. Note that the symbolic superscripts denote whether the visible units corresponds to a feature or a class label.

2. The hidden layer contains m binary units. The hidden units are denoted by $h_1 \ldots h_m$.

The weight of the connection between the ith feature-specific visible unit $v_i^{(f)}$ and the jth hidden unit h_j is given by w_{ij}. This results in a $d \times m$ connection matrix $W = [w_{ij}]$. The weight of the connection between the ith class-specific visible unit $v_i^{(c)}$ and the jth hidden unit h_j is given by u_{ij}. This results in a $k \times m$ connection matrix $U = [u_{ij}]$. The relationships between different types of nodes and matrices for $d = 6$ features, $k = 3$ classes, and $m = 5$ hidden features is shown in Figure 7.6. The bias for the ith feature-specific visible node is denoted by $b_i^{(f)}$, and the bias for the ith class-specific visible node is denoted by $b_i^{(c)}$. The bias for the jth hidden node is denoted by b_j (with no superscript). The states of the hidden nodes are defined in terms of all visible nodes using the sigmoid function:

$$P(h_j = 1|\overline{v}^{(f)}, \overline{v}^{(c)}) = \frac{1}{1 + \exp(-b_j - \sum_{i=1}^{d} v_i^{(f)} w_{ij} - \sum_{i=1}^{k} v_i^{(c)} u_{ij})} \tag{7.23}$$

Note that this is the standard way in which the probabilities of hidden units are defined in a Boltzmann machine. There are, however, some differences between how the probabilities of the feature-specific visible units and the class-specific visible units are defined. In the case of the feature-specific visible units, the relationship is not very different from a standard Boltzmann machine:

$$P(v_i^{(f)} = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(f)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{7.24}$$

The case of the class units is, however, slightly different because we must use the softmax function instead of the sigmoid. This is because of the one-hot encoding of the class. Therefore, we have the following:

$$P(v_i^{(c)} = 1|\overline{h}) = \frac{\exp(b_i^{(c)} + \sum_j h_j u_{ij})}{\sum_{l=1}^{k} \exp(b_l^{(c)} + \sum_j h_j u_{lj})} \tag{7.25}$$

A naive approach to training the Boltzmann machine would use a similar generative model to previous sections. The multinomial model is used to generate the visible states $v_i^{(c)}$ for the classes. The corresponding updates of the contrastive divergence algorithm are as follows:

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i^{(f)}, h_j \rangle_{pos} - \langle v_i^{(f)}, h_j \rangle_{neg} \right) \quad \text{if } i \text{ is feature unit}$$
$$u_{ij} \Leftarrow u_{ij} + \alpha \left(\langle v_i^{(c)}, h_j \rangle_{pos} - \langle v_i^{(c)}, h_j \rangle_{neg} \right) \quad \text{if } i \text{ is class unit}$$

This approach is a direct extension from collaborative filtering. However, the main problem is that this *generative* approach does not fully optimize for classification accuracy. To provide an analogy with autoencoders, one would not necessarily perform significantly better dimensionality reduction (in a supervised sense) by simply including the class variable among the inputs. The reduction would often be dominated by the unsupervised relationships among the features. Rather, the *entire focus* of the learning should be on optimizing the accuracy of classification. Therefore, a *discriminative* approach to training the RBM is often used in which the weights are learned to maximize the conditional class likelihood of the true class. Note that it is easy to set up the conditional probability of the class variable, given the visible states by using the probabilistic dependencies between the hidden features and classes/features. For example, in the traditional form of a restrictive Boltzmann machine, we are maximizing the *joint* probability of the feature variables $v_i^{(f)}$ and the class variables v_i^c. However, in the discriminative variant, the objective function is set up to maximize the *conditional* probability of the class variable $y \in \{1 \ldots k\}$ $P(v_y^{(c)} = 1 | \overline{v}^{(f)})$. Such an approach has a more focused effect of maximizing classification accuracy. Although it is possible to train a discriminative restricted Boltzmann machine using contrastive divergence, the problem is simplified because one can estimate $P(v_y^{(c)} = 1 | \overline{v}^{(f)})$ in closed form without having to use an iterative approach. This form can be shown to be the following [269, 431]:

$$P(v_y^{(c)} = 1 | \overline{v}^{(f)}) = \frac{\exp(b_y^{(c)}) \prod_{j=1}^{m} [1 + \exp(b_j^{(h)} + u_{yj} + \sum_i w_{ij} v_i^{(f)})]}{\sum_{l=1}^{k} \exp(b_l^{(c)}) \prod_{j=1}^{m} [1 + \exp(b_j^{(h)} + u_{lj} + \sum_i w_{ij} v_i^{(f)})]} \tag{7.26}$$

With this differentiable closed form, it is a simple matter to differentiate the negative logarithm of the above expression for stochastic gradient descent. If $\mathcal{L}$ is the negative logarithm of the above expression and θ is any particular parameter (e.g., weight or bias) of the Boltzmann machine, one can show the following:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{j=1}^{m} \text{Sigmoid}(o_{yj}) \frac{\partial o_{yj}}{\partial \theta} - \sum_{l=1}^{k} \sum_{j=1}^{m} \text{Sigmoid}(o_{lj}) \frac{\partial o_{lj}}{\partial \theta} \tag{7.27}$$

Here, we have $o_{yj} = b_j^{(h)} + u_{yj} + \sum_i w_{ij} v_i^{(f)}$. The above expression can be easily computed for each training point and for each parameter in order to perform the stochastic gradient descent process. It is a relatively simple matter to make probabilistic predictions for unseen test instances using Equation 7.26. More details and extensions are discussed in [269].

7.5.4 Topic Models with RBMs

Topic modeling is a form of dimensionality reduction that is specific to text data. The earliest topic models, which correspond to Probabilistic Latent Semantic Analysis (PLSA), were proposed in [216]. In PLSA, the basis vectors are not orthogonal to one another, as

is the case with SVD. On the other hand, both the basis vectors and the transformed representations are constrained to be nonnegative values. The nonnegativity in the value of each transformed feature is semantically useful, because it represents the strength of a topic in a particular document. In the context of the RBM, this strength corresponds to the probability that a particular hidden unit takes on the value of 1, given that the words in a particular document have been observed. Therefore, one can use the vector of conditional probabilities of the hidden states (when visible states are fixed to document words) in order to create a reduced representation of each document. It is assumed that the lexicon size is d, whereas the number of hidden units is $m \ll d$.

This approach shares some similarities with the technique used for collaborative filtering in which a single RBM is created for each user (row of the matrix). In this case, a single RBM is created for each document. A group of visible units is created for each word, and therefore the number of groups of visible units is equal to the number of words in the document. In the following, we will concretely define how the visible and hidden states of the RBM are fixed in order to describe the concrete workings of the model:

1. For the tth document containing n_t words, a total of n_t softmax groups are retained. Each softmax group contains d nodes corresponding to the d words in the lexicon. Therefore, the RBM for each document is different, because the number of units depends on the length of the document. However, all the softmax groups within a document and across multiple documents share weights of their connections to the hidden units. The ith position in the document corresponds to the ith group of visible softmax units. The ith group of visible units is denoted by $v_i^{(1)} \ldots v_i^{(d)}$. The bias associated with $v_i^{(k)}$ is $b^{(k)}$. Note that the bias of the ith visible node depends only on k (word identity) and not on i (position of word in document). This is because the model uses a bag-of-words approach in which the positions of the words are irrelevant.

2. There are m hidden units denoted by $h_1 \ldots h_m$. The bias of the jth hidden unit is b_j.

3. Each hidden unit is connected to each of the $n_t \times d$ visible units. All softmax groups within a single RBM as well as across different RBMs (corresponding to different documents) share the same set of d weights. The kth hidden unit is connected to a group of d softmax units with a vector of d weights denoted by $\overline{W}^{(k)} = (w_1^{(k)} \ldots w_d^{(k)})$. In other words, the kth hidden unit *is connected to each of the n_t groups of d softmax units with the same set of weights* $\overline{W}^{(k)}$.

The architecture of the RBM is illustrated in Figure 7.7. Based on this architecture, one can express the probabilities associated with the hidden states as follows:

$$P(h_j = 1 | \overline{v}^{(1)}, \ldots \overline{v}^{(d)}) = \frac{1}{1 + \exp(-b_j - \sum_{i=1}^{n_t} \sum_{k=1}^{d} v_i^{(k)} w_j^{(k)})} \tag{7.28}$$

One can also express the visible states with the use of the multinomial model:

$$P(v_i^{(k)} = 1 | \overline{h}) = \frac{\exp(b^{(k)} + \sum_{j=1}^{m} w_j^{(k)} h_j)}{\sum_{l=1}^{d} \exp(b^{(l)} + \sum_{j=1}^{m} w_j^{(l)} h_j)} \tag{7.29}$$

The normalization factor in the denominator ensures that for each visible unit the sum of the probabilities over all words is 1. Furthermore, the right-hand side of the above equation

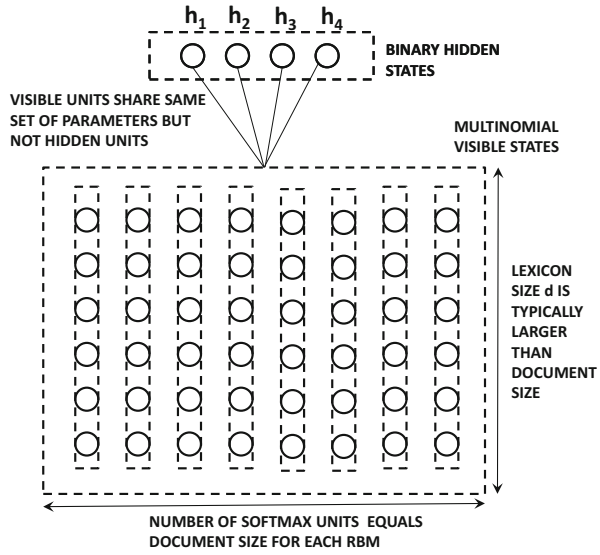


Figure 7.7: The RBM for each document is illustrated. The number of visible units is equal to the number of words in each document

is independent of the index i of the visible unit, because this bag-of-words model does not depend on the position of words in the document.

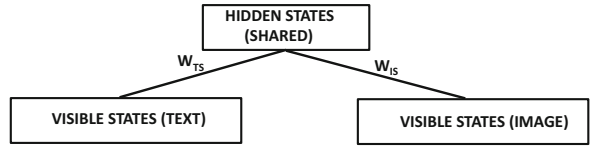
With these relationships, one can apply MCMC sampling to generate samples of the hidden and visible states for the contrastive divergence algorithm. Note that the RBMs are different for different documents, although these RBMs share weights. As in the case of the collaborative filtering application, each RBM is associated with only a single training example corresponding to the relevant document. The weight update used for gradient descent is the same as used for the traditional RBM. The only difference is that the weights across different visible units are shared. This approach is similar to what is performed in collaborative filtering. We leave the derivation of the weight updates as an exercise for the reader (see Exercise 5).

After the training has been performed, the reduced representation of each document is computed by applying Equation 7.28 to the words of a document. The real-valued value of the probabilities of the hidden units provides the m-dimensional reduced representation of the document. The approach described in this section is a simplification of a multilayer approach described in the original work [485].

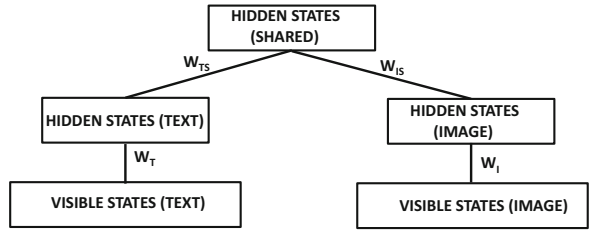
7.5.5 RBMs for Machine Learning with Multimodal Data

Boltzmann machines can also be used for machine learning with *multimodal* data. Multimodal data refers to a setting in which one is trying to extract information from data points with multiple modalities. For example, an image with a text description can be considered multimodal data. This is because this data object has both image and text modalities.

The main challenge in processing multimodal data is that it is often difficult to use machine learning algorithms on such heterogeneous features. Multimodal data is often processed by using a shared representation in which the two modes are mapped into a joint



(a) A simple RBM for multimodal data



(a) A multimodal RBM with an added hidden layer

Figure 7.8: RBM architecture for multimodal data processing

space. A common approach for this goal is *shared* matrix factorization. Numerous methods for using shared matrix factorization with text and image data are discussed in [7]. Since RBMs provide alternative representations to matrix factorization methods in many settings, it is natural to explore whether one can use this architecture to create a shared latent representation of the data.

An example [484] of an architecture for multimodal modeling is shown in Figure 7.8(a). In this example, it is assumed that the two modes correspond to text and image data. The image and the text data are used to create hidden states that are specific to images and text, respectively. These hidden states then feed into a single shared representation. The similarity of this architecture with the classification architecture of Figure 7.6 is striking. This is because both architectures try to map two types of features into a set of shared hidden states. These hidden states can then be used for different types of inference, such as using the shared representation for classification. As shown in section 7.7, one can even enhance such unsupervised representations with backpropagation to fine-tune the approach. Missing data modalities can also be generated using this model.

One can optionally improve the expressive power of this model by using depth. An additional hidden layer has been added between the visible states and the shared representation in Figure 7.8(b). Note that one can add multiple hidden layers in order to create a deep network. However, we have not yet described how one can actually train a multilayer RBM. This issue is discussed in section 7.7.

An additional challenge with the use of multimodal data is that the features are often not binary. There are several solutions to this issue. In the case of text (or data modalities with small cardinalities of discrete attributes), one can use a similar approach as used in the RBM for topic modeling where the count c of a discrete attribute is used to create c instances of the one-hot encoded attribute. The issue becomes more challenging when the data contains arbitrary real values. These issues are discussed in the next section.

7.6 Using RBMs beyond Binary Data Types

The entire discussion in this chapter has focussed on binary data types, which is the setting used by the vast majority of RBMs. For some types of data, such as categorical data or ordinal data (e.g., ratings), one can use the softmax approach described in section 7.5.2. For example, the use of softmax units for word-count data is discussed in section 7.5.4. One can make the softmax approach work with an ordered attribute, when the number of discrete values of that attribute is small.

The approach described in section 7.5.2 does provide some hints about how different data types can be addressed. For example, categorical or ordinal data is handled *by changing the probability distribution* of visible units to be more appropriate to the problem at hand. In general, one might need to change the distribution of not only the visible units, but also the hidden units. This is because the nature of the hidden units is dependent on the visible units.

For real-valued data, a natural solution is to use Gaussian visible units. Furthermore, the hidden units are real-valued as well, and are assumed to contain a ReLU activation function. The energy for a particular combination $(\overline{v}, \overline{h})$ of visible and hidden units is given by the following:

$$E(\overline{v}, \overline{h}) = \underbrace{\sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2}}_{\text{Containment function}} - \sum_j b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij} \tag{7.30}$$

Note that the energy contribution of the bias of visible units is given by a *parabolic containment function*. The effect of using this containment function is to keep the value of the ith visible unit close to b_i. As is the case for other types of Boltzmann machines, the derivatives of the energy function with respect to the different variables also provide the derivatives of the log-likelihoods. This is because the probabilities are always defined by exponentiating the energy function.

There are several challenges associated with the use of this approach. An important issue is that the approach is rather unstable with respect to the choice of the variance parameter σ. In particular, updates to the visible layer tend to be too small, whereas updates to the hidden layer tend to be too large. One natural solution to this dilemma is to use more hidden units than visible units. It is also common to normalize the input data to unit variance so that the standard deviation σ of the visible units can be set to 1. The ReLU units are modified to create a noisy version. Specifically, Gaussian noise with zero mean and variance $\log(1+\exp(v))$ is added to the value of the unit before thresholding it to nonnegative values. The motivation behind using such an unusual activation function is that it can be shown to be equivalent to a *binomial unit* [359, 512], which encodes more information than the binary unit that is normally used. It is important to enable this ability when working with real-valued data. The Gibbs sampling of the real-valued RBM is similar to a binary RBM, as are the updates to the weights once the MCMC samples are generated. It is important to keep the learning rates low to prevent instability.

7.7 Stacking Restricted Boltzmann Machines

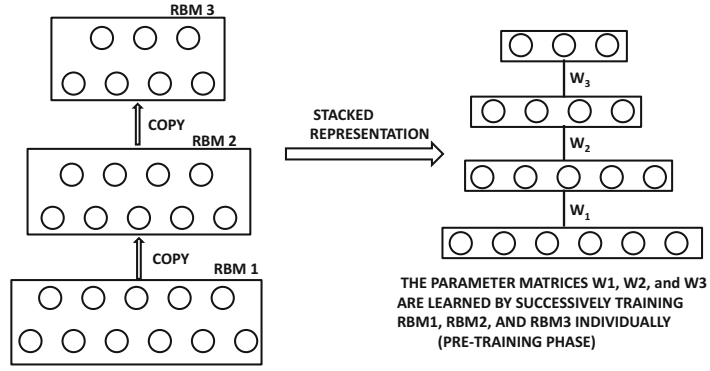
Most of the power of conventional neural architectures arises from having multiple layers of units. Deeper networks are known to be more powerful, and can model more complex

functions at the expense of fewer parameters. A natural question arises concerning whether similar goals can be achieved by putting together multiple RBMs. It turns out that the RBM is well suited to creating deep networks, and was used *earlier* than conventional neural networks for creating deep models with pretraining. In other words, the RBM is trained with Gibbs sampling, and the resulting weights are grandfathered into a conventional neural network with continuous sigmoid activations (instead of sigmoid-based discrete sampling). Why should one go through the trouble to train an RBM in order to train a conventional network at all? This is because of the fact that Boltzmann machines are trained in a fundamentally different way from the backpropagation approach in conventional neural networks. The contrastive divergence approach tends to train all layers jointly, which does not cause the same problems with the vanishing and exploding gradient problems, as is the case in conventional neural networks.

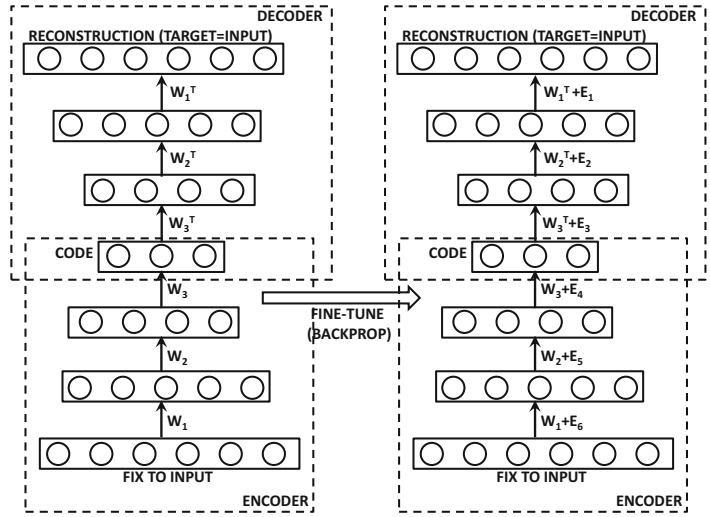
At first glance, the goal of creating deep networks from RBMs seems rather difficult. First, RBMs are not quite like feed-forward units that perform the computation in a particular direction. RBMs are symmetric models in which the visible units and hidden units are connected in the form of an undirected graphical model. Therefore, one needs to define a concrete way in which multiple RBMs interact with one another. In this context, a useful observation is that even though RBMs are symmetric and discrete models, the learned weights can be used to define a related neural network that performs directed computation in the continuous space of activations. These weights are already quite close to the final solution because of how they have been learned with discrete sampling. Therefore, these weights can be fine-tuned with a relatively modest effort of traditional backpropagation. In order to understand this point, consider the single-layer RBM illustrated in Figure 7.4, which shows that even the single-layer RBM is equivalent to a directed graphical model of infinite length. However, once the visible states have been fixed, it suffices to keep only three layers of this computational graph, and perform the computations with the continuous values derived from sigmoid activations. This approach already provides a good approximate solution. The resulting network is a traditional autoencoder, although its weights have been (approximately) learned in a rather unconventional way. This section will show how this type of approach can also be applied to stacked RBMs.

What is a stacked set of RBMs? Consider a data set with d dimensions, for which the goal is to create a reduced representation with m_1 dimensions. One can achieve this goal with an RBM containing d visible units and m_1 hidden units. By training this RBM, one will obtain an m_1-dimensional representation of the data set. Now consider a second RBM that has m_1 visible units and m_2 hidden units. We can simply *copy* the m_1 outputs of the first RBM as the inputs to the second RBM, which has $m_1 \times m_2$ weights. As a result, one can train this new RBM to create an m_2-dimensional representation by using the outputs from the first RBM as its inputs. Note that we can repeat this process for k times, so that the last RBM is of size $m_{k-1} \times m_k$. Therefore, we *sequentially* train each of these RBMs by copying the output of one RBM into the input of another.

An example of a stacked RBM is shown on the left-hand side of Figure 7.9(a). This type of RBM is often shown with the concise diagram on the right-hand side of Figure 7.9(a). Note that the copying between two RBMs is a simple one-to-one copying between corresponding nodes, because the output layer of the rth RBM has exactly the same number of nodes as the input layer of the $(r+1)$th RBM. The resulting representations are *unsupervised* because they do not depend on a specific target. Another point is that the Boltzmann machine is an undirected model. However, by stacking the Boltzmann machine, we no longer have an undirected model because the upper layers receive feedback from the lower layers, but not vice versa. In fact, one can treat each Boltzmann machine as a single computational



(a) The stacked RBMs are trained sequentially in pretraining



(b) Pretraining is followed by fine-tuning with backpropagation

Figure 7.9: Training a multi-layer RBM

unit with many inputs and outputs, and the copying from one machine to another as the data transmission between two computational units. From this particular view of the stack of Boltzmann machines as a computational graph, it is even possible to perform backpropagation if one reverts to using the sigmoid units to create real-valued activations rather than to create the parameters needed for drawing binary samples. Although the use of real-valued activations is only an approximation, it already provides an excellent approximation because of the way in which the Boltzmann machine has been trained. This initial set of weights can be fine-tuned with backpropagation. After all, backpropagation can be performed on any computational graph, irrespective of the nature of the function computed inside the graph as long as a continuous function is computed. The fine tuning of backpropagation approach is particularly essential in the case of supervised learning, because the weights learned from a Boltzmann machine are always unsupervised.

7.7.1 Unsupervised Learning

Even in the case of unsupervised learning, the stacked RBM will generally provide reductions of better quality than a single RBM. However, the training of this RBM has to be performed carefully because results of high quality are not obtained by simply training all the layers together. Better results are obtained by using a pretraining approach. Each of the three RBMs in Figure 7.9(a) are trained sequentially. First, RBM1 is trained using the provided training data as the values of the visible units. Then, the outputs of the first RBM are used to train RBM2. This approach is repeated to train RBM3. Note that one can greedily train as many layers as desired using this approach. Assume that the weight matrices for the three learned RBMs are W_1, W_2, and W_3, respectively. Once these weight matrices have been learned, one can put together an encoder-decoder pair with these three weight matrices as shown in Figure 7.9(b). The three decoders have weight matrices W_1^T, W_2^T, and W_3^T, because they perform the inverse operations of encoders. As a result, one now has a directed encoder-decoder network that can be trained with backpropagation like any conventional neural network. The states in this network are computed using directed probabilistic operations, rather than sampled with the use of Monte-Carlo methods. One can perform backpropagation through the layers in order to fine-tune the learning. Note that the weight matrices on the right-hand side of Figure 7.9(b) have been adjusted as a result of this fine tuning. Furthermore, the weight matrices of the encoder and the decoder are no longer related in a symmetric way as a result of the fine tuning. Such stacked RBMs provide reductions of higher quality compared to those with shallower RBMs [431], which is analogous to the behavior of conventional neural networks.

7.7.2 Supervised Learning

How can one learn the weights in such a way that the Boltzmann machine is encouraged to produce a particular type of output such a class labels? Imagine that one wants to perform a k-way classification with a stack of RBMs. The use of a single-layer RBM for classification has already been discussed in section 7.5.3, and the corresponding architecture is illustrated in Figure 7.6. This architecture can be modified by replacing the single hidden layer with a stack of hidden layers. The final layer of hidden features are then connected to the visible softmax layer that outputs the k probabilities corresponding to the different classes. As in the case of dimensionality reduction, pretraining is helpful. Therefore, the first phase is completely unsupervised in which the class labels are not used. In other words, we train the weights of each hidden layer separately. This is achieved by training the weights of the lower layers first and then the higher layers, as in any stacked RBM. After the initial weights have been set in an unsupervised way, one can perform the initial training of weights between the final hidden layer and visible layer of softmax units. One can then create a directed computational graph with these initial weights, as in the case of the unsupervised setting. Backpropagation is performed on this computational graph in order to perform fine tuning of the learned weights.

7.7.3 Deep Boltzmann Machines and Deep Belief Networks

One can stack the different layers of the RBM in various ways to achieve different types of goals. In some forms of stacking, the interactions between different Boltzmann machines are bi-directional. This variation is referred to as a *deep Boltzmann machine*. In other forms of stacking, some of the layers are uni-directional, whereas others are bi-directional. An

example is a *deep belief network* in which only the upper RBM is bi-directional, whereas the lower layers are uni-directional. Some of these methods can be shown to be equivalent to various types of probabilistic graphical models like *sigmoid belief nets* [361].

A deep Boltzmann machine is particularly noteworthy because of the bi-directional connections between each pair of units. The fact that the copying occurs both ways means that we can merge the nodes in adjacent nodes of two RBMs into a single layer of nodes. Furthermore, observe that one could rearrange the RBM into a bipartite graph by putting all the odd layers in one set and the even layers in another set. In other words, the deep RBM is equivalent to a single RBM. The difference from a single RBM is that the visible units form only a small subset of the units in one layer, and all pairs of nodes are not connected. Because of the fact that all pairs of nodes are not connected, the nodes in the upper layers tend to receive smaller weights than the nodes in the lower layers. As a result, pretraining again becomes necessary in which the lower layers are trained first, and then followed up with the higher layers in a greedy way. Subsequently, all layers are trained together in order to fine-tune the method. Refer to the bibliographic notes for details of these advanced models.

7.8 Summary

The earliest variant of the Boltzmann machine was the Hopfield network. The Hopfield network is an energy-based model, which stores the training data instances in its local minima. The Hopfield network can be trained with the Hebbian learning rule. A stochastic variant of the Hopfield network is the Boltzmann machine, which uses a probabilistic model to achieve greater generalization. Furthermore, the hidden states of the Boltzmann machine hold a reduced representation of the data. The Boltzmann machine can be trained with a stochastic variant of the Hebbian learning rule. The main challenge in the case of the Boltzmann machine is that it requires Gibbs sampling, which can be slow in practice. The restricted Boltzmann machine allows connections only between hidden nodes and visible nodes, which eases the training process. More efficient training algorithms are available for the restricted Boltzmann machine. The restricted Boltzmann machine can be used as a dimensionality reduction method; it can also be used in recommender systems with incomplete data. The restricted Boltzmann machine has also been generalized to count data, ordinal data, and real-valued data. However, the vast majority of RBMs are still constructed under the assumption of binary units. In recent years, several deep variants of the restricted Boltzmann machine have been proposed, which can be used for conventional machine learning applications like classification.

7.9 Bibliographic Notes and Software Resources

The earliest variant of the Boltzmann family of models was the Hopfield network [217]. The Storkey learning rule is proposed in [487]. The earliest algorithms for learning Boltzmann machines with the use of Monte Carlo sampling were proposed in [1, 207]. Discussions of Markov Chain Monte Carlo methods are provided in [144, 362], and many of these methods are useful for Boltzmann machines as well. RBMs were originally invented by Smolensky, and referred to as the harmonium. A tutorial on energy-based models is provided in [286]. Boltzmann machines are hard to train because of the interdependent stochastic nature of the units. The intractability of the partition function

also makes the learning of the Boltzmann machine hard. However, one can estimate the partition function with *annealed importance sampling* [363]. A variant of the Boltzmann machine is the *mean-field Boltzmann machine* [388], which uses deterministic real units rather than stochastic units. However, the approach is a heuristic and hard to justify. Nevertheless, the use of real-valued approximations is popular at inference time. In other words, a traditional neural network with real-valued activations and derived weights from the trained Boltzmann machine is often used for prediction. Other variations of the RBM, such as the neural autoregressive distribution estimator [271], can be viewed as autoencoders.

The efficient mini-batch algorithm for Boltzmann machines is described in [508]. The contrastive divergence algorithm, which is useful for RBMs, is described in [61, 201]. A variation referred to as *persistent contrastive divergence* is proposed in [508]. The idea of gradually increasing the value of k in CD_k over the progress of training was proposed in [61]. The work in [61] showed that even a single iteration of the Gibbs sampling approach (which greatly reduces burn-in time) produces only a small bias in the final result, which can be reduced by gradually increasing the value of k in CD_k over the course of training. This insight was key to the efficient implementation of the RBM. An analysis of the bias in the contrastive divergence algorithm may be found in [30]. The work in [495] analyzes the convergence properties of the RBM. It is also shows that the contrastive divergence algorithm is a heuristic, which does not really optimize any objective function. A discussion and practical suggestions for training Boltzmann machines may be found in [124, 203]. The universal approximation property of RBMs is discussed in [352].

RBMs have been used for a variety of applications like dimensionality reduction, collaborative filtering, topic modeling and classification. The use of the RBM for collaborative filtering is discussed in [431]. This approach is instructive because it also shows how one can use an RBM for categorical data containing a small number of values. The application of discriminative restricted Boltzmann machines to classification is discussed in [269, 270]. The topic modeling of documents with Boltzmann machines with softmax units (as discussed in the chapter) is based on [485]. Advanced RBMs for topic modeling with a Poisson distribution are discussed in [140, 560]. The main problem with these methods is that they are unable to work well with documents of varying lengths. The use of replicated softmax is discussed in [209]. This approach is closely connected to ideas from *semantic hashing* [432].

Most of the RBMs are proposed for binary data. However, in recent years, RBMs have also been generalized to other data types. The modeling of count data with softmax units is discussed in the context of topic modeling in [485]. The challenges associated with this type of modeling are discussed in [89]. The use of the RBM for the exponential distribution family is discussed in [541], and discussion for real-valued data is provided in [359]. The introduction of binomial units to encode more information than binary units was proposed in [512]. This approach was shown to be a noisy version of the ReLU [359]. The replacement of binary units with linear units containing Gaussian noise was first proposed in [129]. The modeling of documents with deep Boltzmann machines is discussed in [485]. Boltzmann machines have also been used for multimodal learning with images and text [368, 484].

Training of deep variations of Boltzmann machines provided the first deep learning algorithms that worked well [206]. These algorithms were the first pretraining methods, which were later generalized to other types of neural networks. A detailed discussion of pretraining may be found in section 5.7 of Chapter 5. Deep Boltzmann machines are discussed in [434], and efficient algorithms are discussed in [210, 435].

Several architectures that are related to the Boltzmann machine provide different types of modeling capabilities. The Helmholtz machine and a wake-sleep algorithm are proposed in [205]. RBMs and their multilayer variants can be shown to be equivalent to different types of probabilistic graphical models such as sigmoid belief nets [361]. A detailed discussion of probabilistic graphical models may be found in [259]. In higher-order Boltzmann machines, the energy function is defined by groups of k nodes for $k > 2$. For example, an order-3 Boltzmann machine will contain terms of the form $w_{ijk}s_i s_j s_k$. Such higher-order machines are discussed in [457]. Although these methods are potentially more powerful that traditional Boltzmann machines, they have not found much popularity because of the large amount of data they require to train.

7.10 Exercises

1. This chapter discusses how Boltzmann machines can be used for collaborative filtering. Even though discrete sampling of the contrastive divergence algorithm is used for learning the model, the final phase of inference is done using real-valued sigmoid and softmax activations. Discuss how you can use this fact to your advantage in order to fine-tune the learned model with backpropagation.

2. Implement the contrastive divergence algorithm of a restricted Boltzmann machine. Also implement the inference algorithm for deriving the probability distribution of the hidden units for a given test example. Use Python or any other programming language of your choice.

3. Propose an approach for using RBMs for outlier detection.

4. Derive the weight updates for the RBM-based topic modeling approach discussed in the chapter. Use the same notations.

5. Show how you can extend the RBM for collaborative filtering (discussed in section 7.5.2 of the chapter) with additional layers to make it more powerful.

6. A discriminative Boltzmann machine is introduced for classification towards the end of section 7.5.3. However, this approach is designed for binary classification. Show how you can extend the approach to multi-way classification.

7. Show how you can modify the topic modeling RBM discussed in the chapter in order to create a hidden representation of each node drawn from a large, sparse graph (like a social network).

8. Discuss how you can enhance the model of Exercise 7 to include data about an unordered list of keywords associated with each node. (For example, social network nodes are associated with wall-post and messaging content.)

9. Discuss how you can enhance the topic modeling RBM discussed in the chapter with multiple layers.

Chapter 8

Recurrent Neural Networks

> "Democracy is the recurrent suspicion that more than half the people are right more than half the time."– *The New Yorker*, July 3, 1944.

8.1 Introduction

All the neural architectures discussed in earlier chapters are inherently designed for multidimensional data in which there is no inherent ordering among attributes and the number of dimensions (input data items) are fixed. However, sequential data types such as time-series, text, and biological data do not obey this characteristic:

1. In a time-series, the sequence of values is ordered. Permuting the temporal order of values completely loses the signal in the time series. A key point is that the value of a time-series at time t is closely related to its values in the previous window.

2. Although text is often processed as a bag of words, one can obtain better semantic insights from sentences by using the sequential ordering of words. It is noteworthy that the lengths of sentences may vary quite a bit.

3. Biological data often contains sequences, in which the symbols might correspond to amino acids or one of the nucleobases that form the building blocks of DNA.

The individual values in a sequence can be either real-valued or symbolic. Real-valued sequences are also referred to as time-series. Recurrent neural networks can be used for either type of data, although the most common use-case occurs in text sequences, which are symbolic.

Sequence-centric applications like text are sometimes processed as bags of words. Such an approach ignores the ordering of words in the document, and works well for certain types of applications like classification into broad topics. However, in applications where

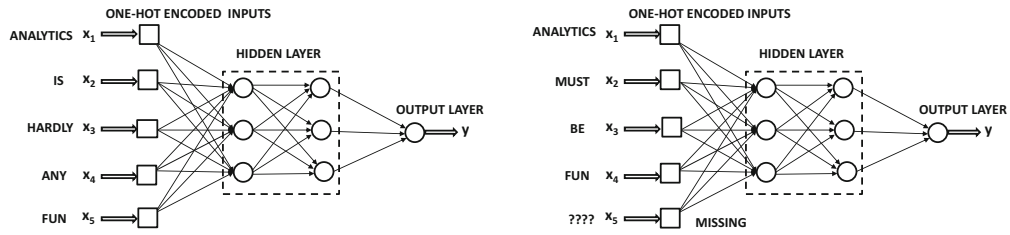
C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_8

the semantic interpretation of the sentence is important, such an approach is inadequate. Consider the following pair of sentences:

The lion chased the deer.
The deer chased the lion.

The two sentences are clearly very different (and the second one is unusual). However, the bag-of-words representation would deem them identical. Hence, this type of representation works well for simpler applications (such as classification), but a greater degree of linguistic intelligence is required for more sophisticated applications such as *sentiment analysis* or *machine translation*.

One possible solution is to avoid the bag-of-words approach and fix the input neuron index for each position in the sequence starting at the sequence beginning. Consider a setting in which this conventional neural network (with one input for each position in the sentence) is used to perform sentiment classification. The sentiment can be a binary label depending on whether it is positive or negative. The first problem that one would face is that the length of different sentences is different. Therefore, if we used a neural network with 5 sets of one-hot encoded word inputs (cf. Figure 8.1(a)), it would be impossible to enter a sentence with more than five words. Furthermore, any sentence with less than five words would have missing inputs (cf. Figure 8.1(b)), and would need padding with dummy words. In some cases, such as Web log sequences, the length of the input sequence might run into the hundreds of thousands, which increases the parameter footprint. More importantly, each word is related to the preceding segment of the sentence with the same grammatical rules, and the architecture does not take this into account. Therefore, *it is important to somehow encode information about the word ordering more directly within the architecture of the network*. The goal of such an approach would be to reduce the parameter requirements with increasing sequence length; recurrent neural networks provide an excellent example of (parameter-wise) *frugal architectural design* with the help of domain-specific insights. Therefore, the two main desiderata for the processing of sequences include (i) the ability to receive and process inputs in the same order as they are present in the sequence, and (ii) the treatment of inputs at each time-stamp in a similar manner in relation to previous history of inputs. A key challenge is that we somehow need to construct a neural network with a fixed number of parameters, but with the ability to process a variable number of inputs.



(a) 5-word sentence
"*Analytics is hardly any fun.*"

(b) 4-word sentence
"*Analytics must be fun.*"

Figure 8.1: An attempt to use a conventional neural network for sentiment analysis faces the challenge of variable-length inputs. The network architecture also does not contain any helpful information about sequential dependencies among successive words.

These desiderata are naturally satisfied with the use of *recurrent neural networks (RNNs)*. In a recurrent neural network, there is a one-to-one correspondence between the layers in the network and the specific positions in the sequence. The position in the sequence is also referred to as its *time-stamp*. Therefore, instead of a variable number of inputs in a single input layer, the network contains a variable number of layers, and each layer has a single input corresponding to that time-stamp. The inputs also are allowed to directly interact with down-stream hidden layers depending on their positions in the sequence. Each layer uses the same set of parameters to ensure similar modeling at each time stamp, and therefore the number of parameters is fixed as well. In other words, the same layer-wise architecture is repeated in time, and therefore the network is referred to as *recurrent*. Recurrent neural networks are also feed-forward networks with a specific structure based on the notion of *time layering*, so that they can take a *sequence* of inputs and produce a sequence of outputs. Each temporal layer can take in an input data point (either single attribute or multiple attributes), and optionally produce a multidimensional output. Such models are particularly useful for sequence-to-sequence learning applications like machine translation or for predicting the next element in a sequence. Recurrent neural networks have been used in the context of many applications like sequence-to-sequence learning, image captioning, machine translation, and sentiment analysis. This chapter will also study the use of recurrent neural networks in the context of these different applications.

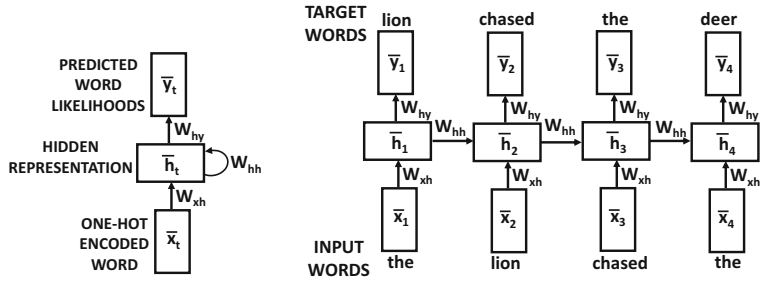
Recurrent neural networks are known to be *Turing complete* [464]. Turing completeness means that a recurrent neural network can simulate any algorithm, given enough data and computational resources [464]. In fact, an architecture known as the neural Turing machine, is inspired by this property. Refer to section 12.3 of Chapter 12 for a detailed discussion. The Turing completeness of recurrent networks is, however, not very useful in practice because the amount of data and computational resources required to achieve this goal can be unrealistic. Furthermore, there are practical issues in training a recurrent neural network, such as the vanishing and exploding gradient problems. These problems increase with the length of the sequence, and more stable variations of the architecture can address this issue only in a limited way.

Chapter Organization

This chapter is organized as follows. The next section will introduce the basic architecture of the recurrent neural network along with the associated training algorithm. The challenges of training recurrent networks are discussed in section 8.3. Because of these challenges, several variations of the recurrent neural network architecture have been proposed. This chapter will study several such variations. Echo-state networks are introduced in section 8.4. Long short-term memory networks are discussed in section 8.5. The gated recurrent unit is discussed in section 8.6. Applications of recurrent neural networks are discussed in section 8.7. A summary is given in section 8.8.

8.2 The Architecture of Recurrent Neural Networks

Although the recurrent neural network can be used in almost any sequential domain, its use in the text domain is both widespread and natural. We will assume the use of the text domain throughout this section in order to enable intuitively simple explanations of various concepts. Therefore, the focus of this chapter will be mostly on discrete RNNs, since that is the most popular use case. It is assumed that the base symbols used to describe a text sequence correspond to the words in the lexicon.



(a) RNN (b) Time-layered representation of (a)

Figure 8.2: A recurrent neural network and its time-layered representation

The simplest recurrent neural network is shown in Figure 8.2(a). A key point here is the presence of the self-loop in Figure 8.2(a), which will cause the hidden state of the neural network to change after the input of each word in the sequence. The architecture of this network becomes clearer by unfolding the loop into a "time-layered" network that looks more like a feed-forward network. This network is shown in Figure 8.2(b). In this case, we have a different node for the hidden state at each time-stamp and the self-loop has been unfurled into a feed-forward network. The input to the ith time-layer node is the ith word. This representation is mathematically equivalent to Figure 8.2(a), but is much easier to comprehend because of its similarity to a traditional network. The weight matrices in different temporal layers *are shared* to ensure that the same function is used at each time-stamp. The annotations W_{xh}, W_{hh}, and W_{hy} of the weight matrices in Figure 8.2(b) make the sharing evident.

It is noteworthy that Figure 8.2 shows a case in which each time-stamp has an input, output, and hidden unit. In practice, it is possible for either the input or the output units to be missing at any particular time-stamp. Examples of cases with missing inputs and outputs are shown in Figure 8.3. The choice of missing inputs and outputs would depend on the specific application at hand. For example, in a time-series forecasting application, we might need outputs at each time-stamp in order to predict the next value in the time-series. On the other hand, in a sequence-classification application, we might only need a single output label at the end of the sequence corresponding to its class. In general, it is possible for any subset of inputs or outputs to be missing in a particular application. The following discussion will assume that all inputs and outputs are present, although it is easy to generalize it to the case where some of them are missing by simply removing the corresponding terms in the forward propagation equations.

The particular architecture shown in Figure 8.2 is suited to *language modeling*. A language model is a well-known concept in natural language processing that predicts the next word, given the previous history of words. The one-hot encoded sequence of words is fed in proper order to the neural network in Figure 8.2(a). This temporal process is equivalent to feeding the individual words to the inputs at the relevant time-stamps (i.e., sequence position index) in Figure 8.2(b). In the setting of language modeling, the output is a vector of probabilities predicted for the next word in the sequence. For example, consider the sentence:

The lion chased the deer.

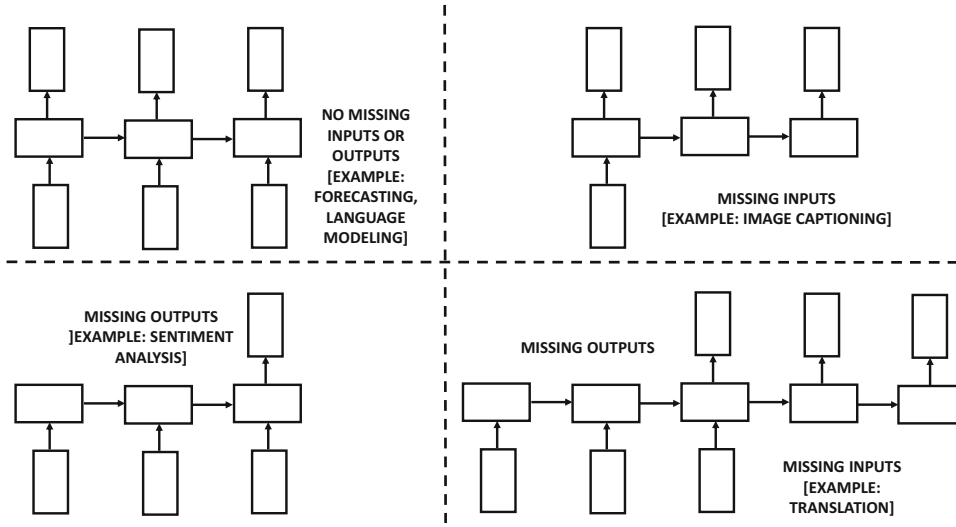


Figure 8.3: The different variations of recurrent networks with missing inputs and outputs

When the word "*The*" is input, the output will be a vector of probabilities of the entire lexicon that includes the word "*lion*," and when the word "*lion*" is input, we will again get a vector of probabilities predicting the next word. This is, of course, the classical definition of a language model in which the probability of a word is estimated based on the immediate history of previous words. In practice, the input is padded on either side with <START> and <END> tokens to demarcate the beginning and end of the input. In general, the input vector at time t (e.g., one-hot encoded vector of the tth word) is $\overline{x}_t$, the hidden state at time t is $\overline{h}_t$, and the output vector at time t (e.g., predicted probabilities of the $(t+1)$th word) is $\overline{y}_t$. Both $\overline{x}_t$ and $\overline{y}_t$ are d-dimensional for a lexicon of size d. The hidden vector $\overline{h}_t$ is p-dimensional, where p regulates the complexity of the embedding. For the purpose of discussion, we will assume that all these vectors are column vectors. In many applications like classification, the output is not produced at each time unit but is only triggered at the last time-stamp in the end of the sentence. Although output and input units may be present only at a subset of the time-stamps, we examine the simple case in which they are present in all time-stamps. Then, the hidden state at time t is given by a function of the input vector at time t and the hidden vector at time $(t-1)$:

$$\overline{h}_t = f(\overline{h}_{t-1}, \overline{x}_t) \tag{8.1}$$

This function is defined with the use of weight matrices and activation functions (as used by all neural networks for learning), and *the same weights are used at each time-stamp*. Therefore, even though the hidden state evolves over time, the weights and the underlying function $f(\cdot, \cdot)$ remain fixed over all time-stamps (i.e., sequential elements) after the neural network has been trained. A separate function $\overline{y}_t = g(\overline{h}_t)$ is used to learn the output probabilities from the hidden states.

Next, we describe the functions $f(\cdot, \cdot)$ and $g(\cdot)$ more concretely. We define a $p \times d$ input-hidden matrix W_{xh}, a $p \times p$ hidden-hidden matrix W_{hh}, and a $d \times p$ hidden-output matrix W_{hy}. Then, one can expand Equation 8.1 and also write the condition for the outputs as follows:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1})$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

Here, the "tanh" notation is defined in vector form, where the function is applied to the p-dimensional column vector in an element-wise fashion to create a p-dimensional vector with each element in $[-1, 1]$. Throughout this section, this vectored notation will be used for activation functions. In the very first time-stamp, $\overline{h}_{t-1}$ is assumed to be some default constant vector (such as 0), because there is no input from the hidden layer at the beginning of a sentence. One can also learn this vector, if desired. Although the hidden states change at each time-stamp, the weight matrices stay fixed over the various time-stamps. Note that the output vector $\overline{y}_t$ is a set of continuous values with the same dimensionality as the lexicon. A softmax layer is applied on top of $\overline{y}_t$ so that the results can be interpreted as probabilities. *The p-dimensional output $\overline{h}_t$ of the hidden layer at the end of a text segment of t words yields its embedding, and the p-dimensional columns of W_{xh} yield the embeddings of individual words.* The latter provides an alternative to *word2vec* embeddings (cf. Chapter 3). In fact, if the training data set is not very large, it is recommended to fix the weights of this layer to pre-trained *word2vec* encodings (in order to avoid overfitting).

Because of the recursive nature of Equation 8.1, the recurrent network has the *ability to compute a function of variable-length inputs*. In other words, one can expand the recurrence of Equation 8.1 to define the function for $\overline{h}_t$ in terms of t inputs. For example, starting at $\overline{h}_0$, which is typically fixed to some constant vector (such as the zero vector), we have $\overline{h}_1 = f(\overline{h}_0, \overline{x}_1)$ and $\overline{h}_2 = f(f(\overline{h}_0, \overline{x}_1), \overline{x}_2)$. Note that $\overline{h}_1$ is a function of only $\overline{x}_1$, whereas $\overline{h}_2$ is a function of both $\overline{x}_1$ and $\overline{x}_2$. In general, $\overline{h}_t$ is a function of $\overline{x}_1 \ldots \overline{x}_t$. Since the output $\overline{y}_t$ is a function of $\overline{h}_t$, these properties are inherited by $\overline{y}_t$ as well. In general, we can write the following:

$$\overline{y}_t = F_t(\overline{x}_1, \overline{x}_2, \ldots \overline{x}_t) \tag{8.2}$$

Note that the function $F_t(\cdot)$ varies with the value of t although its relationship to its immediately previous state is always the same (based on Equation 8.1). Such an approach is particularly useful for variable-length inputs. This setting occurs often in many domains like text in which the sentences are of variable length. For example, in a language modeling application, the function $F_t(\cdot)$ indicates the probability of the next word, taking into account all the previous words in the sentence.

8.2.1 Language Modeling Example of RNN

In order to illustrate the workings of the RNN, we will use a toy example of a single sequence defined on a vocabulary of four words. Consider the sentence:

The lion chased the deer.

In this case, we have a lexicon of four words, which are { "*the*," "*lion*," "*chased*," "*deer*" }. In Figure 8.4, we have shown the probabilistic prediction of the next word at each of time-stamps from 1 to 4. Ideally, we would like the probability of the next word to be predicted correctly from the probabilities of the previous words. Each one-hot encoded input vector $\overline{x}_t$ has length four, in which only one bit is 1 and the remaining bits are 0s. The main flexibility here is in the dimensionality p of the hidden representation, which we set to 2 in this case. As a result, the matrix W_{xh} will be a 2×4 matrix, so that it maps a one-hot encoded input vector into a hidden vector $\overline{h}_t$ vector of size 2. As a practical matter, each column of W_{xh} corresponds to one of the four words, and one of these columns is copied by

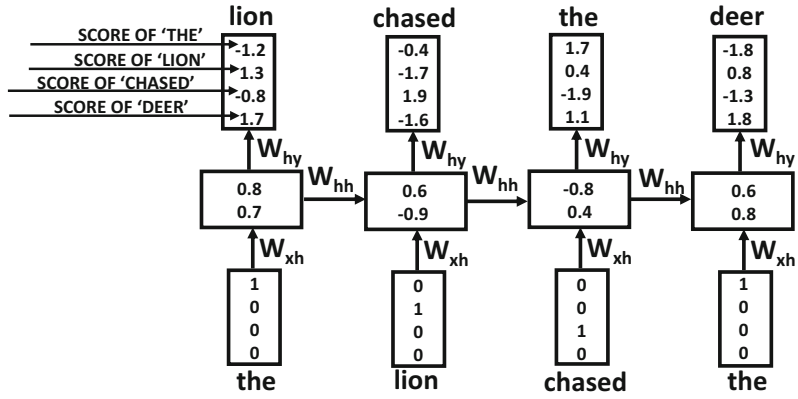


Figure 8.4: Example of language modeling with a recurrent neural network

the expression $W_{xh}\overline{x}_t$. Note that this expression is added to $W_{hh}\overline{h}_t$ and then transformed with the tanh function to produce the final expression. The final output $\overline{y}_t$ is defined by $W_{hy}\overline{h}_t$. Note that the matrices W_{hh} and W_{hy} are of sizes 2×2 and 4×2, respectively.

In this case, the outputs are continuous values (not probabilities) in which larger values indicate greater likelihood of presence. These continuous values are eventually converted to probabilities with the softmax function, and therefore one can treat them as substitutes to log probabilities. The word "*lion*" is predicted in the first time-stamp with a value of 1.3, although this value seems to be (incorrectly) outstripped by "*deer*" for which the corresponding value is 1.7. However, the word "*chased*" seems to be predicted correctly at the next time-stamp. As in all learning algorithms, one cannot hope to predict every value exactly, and such errors are more likely to be made in the early iterations of the backpropagation algorithm. However, as the network is repeatedly trained over multiple iterations, it makes fewer errors over the training data.

Generating a Language Sample

Such an approach can also be used to generate an arbitrary sample of a language, once the training has been completed. How does one use such a language model at *prediction time*, since each state requires an input word, and none is available during language generation (unlike during training)? The likelihoods of the tokens at the first time-stamp can be generated using the <START> token as input. Since the <START> token is also available in the training data, the model will typically select a word that often starts text segments. Subsequently, the idea is to sample one of the tokens generated at each time-stamp (based on the predicted likelihood), and then use it as an input to the next time-stamp. To improve the accuracy of the sequentially predicted token, one might use beam search to expand on the most likely possibilities by always keeping track of the b best sequence prefixes of any particular length. The value of b is a user-driven parameter. By recursively applying this operation, one can generate an arbitrary sequence of text that reflects the particular training data at hand. If the <END> token is predicted, it indicates the end of that particular segment of text. Although such an approach often results in syntactically correct text, it might be nonsensical in meaning. For example, a character-level RNN[1] authored by Karpathy, Johnson, and Fei Fei [243, 604] was trained on William Shakespeare's plays. A

[1] A long-short term memory network (LSTM) was used, which is a variation on the vanilla RNN discussed here. This model is introduced later in this chapter.

character-level RNN requires the neural network to learn both syntax *and* spelling. After only five iterations of learning across the full data set, the following was a sample of the output:

KING RICHARD II:
Do cantant,-'for neight here be with hand her,-
Eptar the home that Valy is thee.

NORONCES:
Most ma-wrow, let himself my hispeasures;
An exmorbackion, gault, do we to do you comforr,
Laughter's leave: mire sucintracce shall have theref-Helt.

Note that there are a large number of misspellings in this case, and a lot of the words are gibberish. However, when the training was continued to 50 iterations, the following was generated as a part of the sample:

KING RICHARD II:
Though they good extremit if you damed;
Made it all their fripts and look of love;
Prince of forces to uncertained in conserve
To thou his power kindless. A brives my knees
In penitence and till away with redoom.

GLOUCESTER:
Between I must abide.

This generated piece of text is largely consistent with the syntax and spelling of the archaic English in William Shakespeare's plays, although there are still some obvious errors. Furthermore, the approach also indents and formats the text in a manner similar to the plays by placing new lines at reasonable locations. Continuing to train for more iterations makes the output almost error-free, and some impressive samples are also available at [245].

Of course, the semantic meaning of the text is limited, and one might wonder about the usefulness of generating such nonsensical pieces of text from the perspective of machine learning applications. The key point here is that by providing an additional *contextual* input, such as the neural representation of an image, the neural network can be made to give intelligent outputs such as a grammatically correct description (i.e., caption) of the image. In other words, language models are best used by generating *conditional* outputs.

The primary goal of the language-modeling RNN is not to create arbitrary sequences of the language, but to provide an architectural base that can be modified in various ways to incorporate the effect of the specific context. For example, applications like machine translation and image captioning learn a language model that is *conditioned* on another input such as a sentence in the source language or an image to be captioned. Therefore, the precise design of the application-dependent RNN will use the same principles as the language-modeling RNN, but will make small changes to this basic architecture in order to incorporate the specific context.

Recurrent neural networks produce two types of useful embeddings:

1. The *activations* of the hidden units at each time-stamp contain the multidimensional embedding of the segment sequence up to that time-stamp. Therefore, a recurrent neural network provides an embedding of each prefix of the sequence.

2. The *weight matrix* W_{xh} from the input to hidden layer contains the embedding of each word. The weight matrix W_{xh} is a $p \times d$ matrix for a lexicon of size d. This means that each of the d columns of this matrix contain a p-dimensional embedding for one of the words. These embeddings provide an alternative to the *word2vec* embeddings discussed in Chapter 3.

For other types of sequential inputs, application-specific embeddings will be learned (such as the embeddings of amino acids in a protein sequence).

8.2.2 Backpropagation Through Time

The negative logarithms of the softmax probability of the correct words at the various time-stamps are aggregated to create the loss function. The softmax function is described in section 2.4.2 of Chapter 2, and we directly use those results here. If the output vector $\overline{y}_t$ can be written as $[\hat{y}_t^1 \ldots \hat{y}_t^d]$, it is first converted into a vector of d probabilities using the softmax function:

$$[\hat{p}_t^1 \ldots \hat{p}_t^d] = \text{Softmax}([\hat{y}_t^1 \ldots \hat{y}_t^d])$$

The softmax function above can be found in Equation 2.17 of Chapter 2. If j_t is the index of the ground-truth word at time t in the training data, then the loss function L for all T time-stamps is computed as follows:

$$L = -\sum_{t=1}^{T} \log(\hat{p}_t^{j_t}) \tag{8.3}$$

This loss function is a direct consequence of Equation 2.18 of Chapter 2. The derivative of the loss function with respect to the raw outputs may be computed as follows (cf. Equation 2.19 of Chapter 2):

$$\frac{\partial L}{\partial \hat{y}_t^k} = \hat{p}_t^k - I(k, j_t) \tag{8.4}$$

Here, $I(k, j_t)$ is an indicator function that is 1 when k and j_t are the same, and 0, otherwise. Starting with this partial derivative, one can use the straightforward backpropagation update of Chapter 2 (on the unfurled temporal network) to compute the gradients with respect to the weights in different layers. However, one also has to account for the sharing of weight matrices in various temporal layers; as discussed in section 2.6.6 of Chapter 2, it is not difficult to modify the backpropagation algorithm to handle shared weights.

The main trick for handling shared weights is to first "pretend" that the parameters in the different temporal layers are independent of one another. For this purpose, we introduce the temporal variables $W_{xh}^{(t)}$, $W_{hh}^{(t)}$ and $W_{hy}^{(t)}$ for time-stamp t. Conventional backpropagation is first performed by working under the pretense that these variables are distinct from one another. Then, the contributions of the different temporal avatars of the weight parameters to the gradient are added to create a unified update for each weight parameter. This special type of backpropagation algorithm is referred to as *backpropagation through time (BPTT)*. We summarize the BPTT algorithm as follows:

(i) Run the input sequentially in the forward direction through time and compute the errors (and the negative-log loss of softmax layer) at each time-stamp.

(ii) Compute the gradients of the edge weights in the backwards direction on the unfurled network without any regard for the fact that weights in different time layers

are shared. In other words, it is assumed that the weights $W_{xh}^{(t)}$, $W_{hh}^{(t)}$ and $W_{hy}^{(t)}$ in time-stamp t are distinct from other time-stamps. As a result, one can use conventional backpropagation to compute $\frac{\partial L}{\partial W_{xh}^{(t)}}$, $\frac{\partial L}{\partial W_{hh}^{(t)}}$, and $\frac{\partial L}{\partial W_{hy}^{(t)}}$. Note that we have used matrix calculus notations where the derivative with respect to a matrix is defined by a corresponding matrix of element-wise derivatives.

(iii) Add all the derivatives with respect to different instantiations of an edge in time as follows:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{xh}^{(t)}}$$
$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hh}^{(t)}}$$
$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hy}^{(t)}}$$

As in all backpropagation methods with shared weights (cf. section 2.6.6 of Chapter 2), we are using the fact that the partial derivative of the loss with respect to each parameter can be obtained by adding the derivative with respect to each copy of the parameter. Furthermore, the computation of the partial derivatives with respect to the temporal copies of each parameter is not different from traditional backpropagation at all. Therefore, one only needs to wrap the temporal aggregation around conventional backpropagation in order to compute the update equations. The original algorithm for backpropagation through time can be credited to Werbos's seminal work in 1990 [545], long before the use of recurrent neural networks became popular.

Truncated Backpropagation Through Time

One of the computational problems in training recurrent networks is that the underlying sequences may be very long, as a result of which the number of layers in the network may also be very large. This can result in computational, convergence, and memory-usage problems. This problem is solved by using *truncated backpropagation through time*. This technique may be viewed as the analog of stochastic gradient descent for recurrent neural networks. In the approach, the state values are computed correctly during forward propagation, but the backpropagation updates are done only over segments of the sequence of modest length (such as 100). In other words, only the portion of the loss over the relevant segment is used to compute the gradients and update the weights. The segments are processed in the same order as they occur in the input sequence. The forward propagation does not need to be performed in a single shot, but it can also be done over the relevant segment of the sequence as long as the values in the final time-layer of the segment are used for computing the state values in the next segment of layers. The values in the final layer in the current segment are used to compute the values in the first layer of the next segment. Therefore, forward propagation is always able to accurately maintain state values, although the backpropagation uses only a small portion of the loss.

Practical Issues

The entries of each weight matrix are initialized to small values in $[-1/\sqrt{r}, 1/\sqrt{r}]$, where r is the number of columns in that matrix. One can also initialize each of the d columns of the input weight matrix W_{xh} to the *word2vec* embedding of the corresponding word (cf. Chapter 3). This approach is a form of pretraining, which is helpful when the amount of training data is small.

Another detail is that the training data often contains a special <START> and an <END> token at the beginning and end of each training segment. These types of tokens help the model to recognize specific text units such as sentences, paragraphs, or the beginning of a particular module of text. The distribution of the words at the beginning of a segment of text is often very different than how it is distributed over the whole training data. Therefore, after the occurrence of <START>, the model is more likely to pick words that begin a particular segment of text. There are other approaches that are used for deciding whether to end a segment at a particular point. A specific example is the use of a binary output that decides whether or not the sequence should continue at a particular point. Note that the binary output is in addition to other application-specific outputs. Such an approach is useful with real-valued sequences.

There are also several practical challenges in training an RNN, which make the design of various architectural enhancements of the RNN necessary. It is also noteworthy that multiple hidden layers (with long short-term memory enhancements) are used in all practical applications, which will be discussed in section 8.2.4. However, the application-centric exposition will use the simpler single-layer model for clarity. The generalization of each of these applications to enhanced architectures is straightforward.

8.2.3 Bidirectional Recurrent Networks

One disadvantage of recurrent networks is that the state at a particular time unit only has knowledge about the past inputs up to a certain point in a sentence, but it has no knowledge about future states. In certain applications like inference of word semantics, the results are vastly improved with knowledge about both past and future states. For example, the specific semantics of the polysemous word "*right*" in "*right choice*" can be inferred only after processing the word "*choice*." Therefore, the correct semantics of a word may be hard to infer without processing positions occurring after the word. Traditional recurrent networks are unable to achieve this goal at least to some extent, because any predictions made by an intermediate state occurring before such informative positions are likely to be lacking in a sufficient amount of context.

Bidirectional recurrent neural networks are appropriate for applications in which the predictions are not causal based on a historical window. A classical example of a causal setting is a stream of symbols in which an event is predicted on the basis of the history of previous symbols. Even though language-modeling applications are formally considered causal applications (i.e., based on immediate history of *previous* words), the reality is that a given word can be predicted with much greater accuracy through the use of the contextual words on each side of it. In general, bidirectional RNNs work well in applications where the predictions are based on bidirectional context. Examples of such applications include handwriting recognition and speech recognition, in which the properties of individual elements in the sequence depend on those on either side of it. For example, if a handwriting is expressed in terms of the strokes, the strokes on either side of a particular position are helpful in recognizing the particular character being synthesized. It has increasingly become

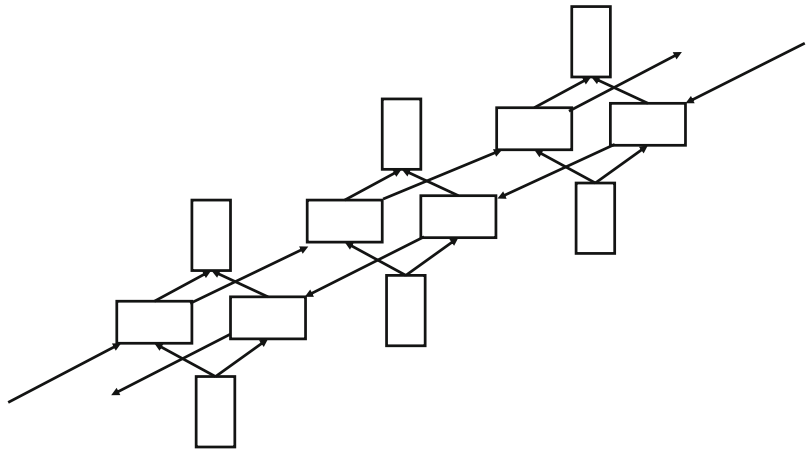


Figure 8.5: Showing three time-layers of a bidirectional recurrent network

common to use bidirectional recurrent networks in various language-centric applications as opposed to unidirectional networks.

In the bidirectional recurrent network, we have separate hidden states $\overline{h}_t^{(f)}$ and $\overline{h}_t^{(b)}$ for the forward and backward directions. The forward hidden states interact only with each other and the same is true for the backward hidden states. The main difference is that the forward states interact in the forwards direction, while the backwards states interact in the backwards direction. Both $\overline{h}_t^{(f)}$ and $\overline{h}_t^{(b)}$, however, receive input from the same vector $\overline{x}_t$ (e.g., one-hot encoding of word) and they interact with the same output vector $\hat{y}_t$. An example of three time-layers of the bidirectional RNN is shown in Figure 8.5.

In the case of the bidirectional network, we have separate forward and backward parameter matrices. The forward matrices for the input-hidden, hidden-hidden, and hidden-output interactions are denoted by $W_{xh}^{(f)}$, $W_{hh}^{(f)}$, and $W_{hy}^{(f)}$, respectively. The backward matrices for the input-hidden, hidden-hidden, and hidden-output interactions are denoted by $W_{xh}^{(b)}$, $W_{hh}^{(b)}$, and $W_{hy}^{(b)}$, respectively.

The recurrence conditions can be written as follows:

$$\begin{aligned}
\overline{h}_t^{(f)} &= \tanh(W_{xh}^{(f)} \overline{x}_t + W_{hh}^{(f)} \overline{h}_{t-1}^{(f)}) \\
\overline{h}_t^{(b)} &= \tanh(W_{xh}^{(b)} \overline{x}_t + W_{hh}^{(b)} \overline{h}_{t+1}^{(b)}) \\
&= \overline{y}_t = W_{hy}^{(f)} \overline{h}_t^{(f)} + W_{hy}^{(b)} \overline{h}_t^{(b)}
\end{aligned}$$

It is easy to see that the bidirectional equations are simple generalizations of the conditions used in a single direction. It is assumed that there are a total of T time-stamps in the neural network shown above, where T is the length of the sequence. One question is about the forward input at the boundary conditions corresponding to $t = 1$ and the backward input at $t = T$, which are not defined. In such cases, one can use a default constant value of 0.5 in each case, although one can also make the determination of these values as a part of the learning process.

An immediate observation about the hidden states in the forward and backwards direction is that they do not interact with one another at all. Therefore, one could first run

the sequence in the forward direction to compute the hidden states in the forward direction, and then run the sequence in the backwards direction to compute the hidden states in the backwards direction. At this point, the output states are computed from the hidden states in the two directions. After the outputs have been computed, the backpropagation algorithm is applied to compute the partial derivatives with respect to various parameters. First, the partial derivatives are computed with respect to the output states because both forward and backwards states point to the output nodes. Then, the backpropagation pass is computed only for the forward hidden states starting from $t = T$ down to $t = 1$. The backpropagation pass is finally computed for the backwards hidden states from $t = 1$ to $t = T$. Finally, the partial derivatives with respect to the shared parameters are aggregated. Therefore, the BPTT algorithm can be generalized to bidirectional networks as follows:

1. Compute forward and backwards hidden states independently in separate passes.
2. Compute output states from backwards and forward hidden states.
3. Compute partial derivatives of loss with respect to output states and each copy of the output parameters.
4. Compute partial derivatives of loss with respect to forward states and backwards states independently using backpropagation. Use these computations to evaluate partial derivatives with respect to each copy of the forwards and backwards parameters.
5. Aggregate partial derivatives over shared parameters.

A bidirectional recurrent network shares some resemblance to an ensemble of two independent recurrent networks — in one, the input is presented in original form, and in the other, the input is reversed. The main difference is that the parameters of the forwards and backwards states are interdependent and therefore trained jointly in bidirectional networks.

8.2.4 Multilayer Recurrent Networks

In all the aforementioned applications, a single-layer RNN architecture is used for ease in understanding. However, in practical applications, a multilayer architecture is used in order to build models of greater complexity. An example of a recurrent network containing three layers is shown in Figure 8.6. Note that nodes in higher-level layers receive input from those in lower-level layers. The relationships among the hidden states can be generalized directly from the single-layer network. First, we rewrite the recurrence equation of the hidden layers (for single-layer networks) in a form that can be adapted easily to multilayer networks:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}) = \tanh\, W \begin{bmatrix} \overline{x}_t \\ \overline{h}_{t-1} \end{bmatrix} \tag{8.5}$$

Here, we have put together a larger matrix $W = [W_{xh}, W_{hh}]$ that includes the columns of W_{xh} and W_{hh}. Similarly, we have created a larger column vector that stacks up the state vector in the first hidden layer at time $t-1$ and the input vector at time t. In order to distinguish between the hidden nodes for the upper-level layers, let us add an additional superscript to the hidden state and denote the vector for the hidden states at time-stamp t and layer k by $\overline{h}_t^{(k)}$. Similarly, let the weight matrix for the kth hidden layer be denoted by $W^{(k)}$. It is noteworthy that the weights are shared across different time-stamps (as in the single-layer recurrent network), but they are not shared across different layers. Therefore,

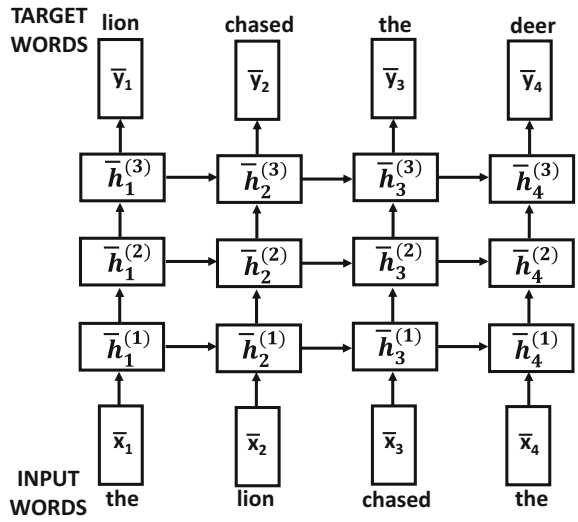


Figure 8.6: Multi-layer recurrent neural networks

the weights are superscripted by the layer index k in $W^{(k)}$. The first hidden layer is special because it receives inputs both from the input layer at the current time-stamp and the adjacent hidden state at the previous time-stamp. Therefore, the matrices $W^{(k)}$ will have a size of $p \times (d+p)$ only for the first layer (i.e., $k = 1$), where d is the size of the input vector $\overline{x}_t$ and p is the size of the hidden vector $\overline{h}_t$. Note that d will typically not be the same as p. The recurrence condition for the first layer is already shown above by setting $W^{(1)} = W$. Therefore, let us focus on all the hidden layers k for $k \geq 2$. The recurrence condition for the layers with $k \geq 2$ is also in a similar form to Equation 8.5:

$$\overline{h}_t^{(k)} = \tanh\, W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix}$$

In this case, the size of the matrix $W^{(k)}$ is $p \times (p+p) = p \times 2p$. The transformation from hidden to output layer remains the same as in single-layer networks. It is common to use two or three layers in practical applications. A larger number of layers will require more training data to avoid overfitting.

8.3 The Challenges of Training Recurrent Networks

Recurrent neural networks are very hard to train because of the fact that the time-layered network can be a very deep network for long input sequences; after all, the depth of the temporal layering is input-dependent. As in all deep networks, the loss function has highly varying magnitudes of the gradients with respect to different temporal layers. Furthermore, the same parameter matrices are shared by different temporal layers. This combination of varying gradient magnitudes and shared parameters in different layers can lead to some unusually unstable effects.

The primary challenges associated with recurrent neural network training are those of the *vanishing* and *exploding gradient problems*. These issues are explained in detail in

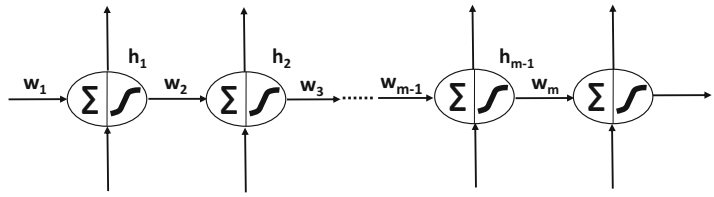


Figure 8.7: The vanishing and exploding gradient problems

section 4.3.2 of Chapter 4. In this section, we will revisit this issue in the context of recurrent neural networks. It is easiest to understand the challenges associated with recurrent networks by examining the case of a recurrent network with a single unit in each layer.

Consider a set of T consecutive layers, in which the tanh activation function, $\Phi(\cdot)$, is applied between each pair of layers. The shared weight between a pair of hidden nodes is denoted by w. Let $h_1 \ldots h_T$ be the hidden values in the various layers. Let $\Phi'(h_t)$ be the derivative of the activation function in hidden layer t. Let the copy of the shared weight w in the tth layer be denoted by w_t so that it is possible to examine the effect of the backpropagation update. Let $\frac{\partial L}{\partial h_t}$ be the derivative of the loss function with respect to the hidden activation h_t. The neural architecture is illustrated in Figure 8.7. Then, one derives the following update equations using backpropagation:

$$\frac{\partial L}{\partial h_t} = \Phi'(w_{t+1} h_t) \cdot w_{t+1} \cdot \frac{\partial L}{\partial h_{t+1}} \tag{8.6}$$

Since the shared weights in different temporal layers are the same, the gradient is multiplied with the same quantity $w_t = w$ for each layer. Such a multiplication will have a consistent bias towards vanishing when $w < 1$, and it will have a consistent bias towards exploding when $w > 1$. However, the choice of the activation function will also play a role because the derivative $\Phi'(w_{t+1} h_t)$ is included in the product. For example, the presence of the tanh activation function, for which the derivative $\Phi'(\cdot)$ is almost always less than 1, tends to increase the chances of the vanishing gradient problem.

Although the above discussion only studies the simple case of a hidden layer with one unit, one can generalize the argument to a hidden layer with multiple units [231]. In such a case, it can be shown that the update to the gradient boils down to a repeated multiplication with the same matrix A. One can show the following result:

Lemma 8.3.1 *Let A be a square matrix, the* **magnitude** *of whose largest eigenvalue is λ. Then, the entries of A^t tend to 0 with increasing values of t, when we have $\lambda < 1$. On the other hand, the entries of A^t diverge to large values, when we have $\lambda > 1$.*

The proof of the above result is easy to show by diagonalizing $A = P\Delta P^{-1}$. Then, it can be shown that $A^t = P\Delta^t P^{-1}$, where Δ is a diagonal matrix. The magnitude of the largest diagonal entry of Δ^t either vanishes with increasing t or it grows to an increasingly large value (in absolute magnitude) depending on whether than eigenvalue is less than 1 or larger than 1. In the former case, the matrix A^t tends to 0, and therefore the gradient vanishes. In the latter case, the gradient explodes. Of course, this does not yet include the effect of the activation function, and one can change the threshold on the largest eigenvalue to set up the conditions for the vanishing or exploding gradients. For example, the largest possible value of the sigmoid activation derivative is 0.25, and therefore the vanishing gradient problem will definitely occur when the largest eigenvalue is less that $1/0.25 = 4$. One can, of course,

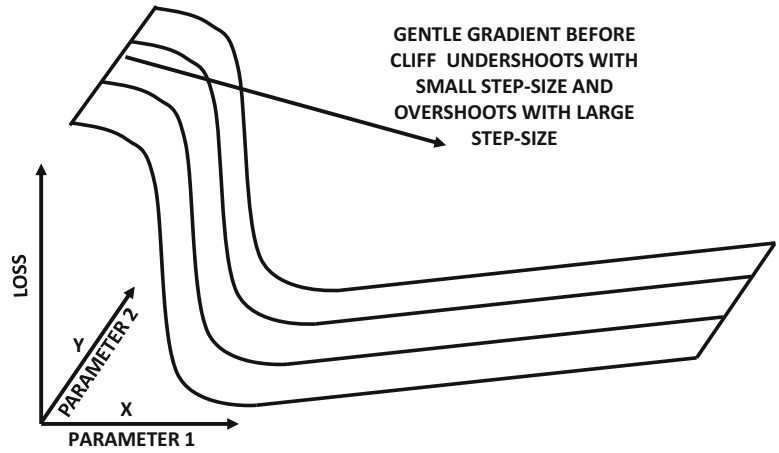


Figure 8.8: Revisiting Figure 4.5 of Chapter 4: An example of a cliff in the loss surface

combine the effect of the matrix multiplication and activation function into a single *Jacobian* matrix (cf. Table 2.1 of Chapter 2), whose eigenvalues can be tested.

In the particular case of recurrent neural networks, the *combination of the vanishing/exploding gradient and the parameter tying across different layers causes the recurrent neural network to behave in an unstable way with gradient-descent step size.* In other words, if we choose a step size that is too small, then the effect of some of the layers will cause little progress. On the other hand, if we choose a step size that is too large, then the effect of some of the layers will cause the step to overshoot the optimal point in an unstable way. An important issue here is that the gradient only tells us the best direction of movement for infinitesimally small steps; for finite steps, the behavior of the update could be substantially different from what is predicted by the gradient. The optimal points in recurrent networks are often hidden near cliffs or other regions of unpredictable change in the topography of the loss function, which causes the best directions of *instantaneous* movement to be extremely poor predictors of the best directions of *finite* movement. Since any practical learning algorithm is required to make finite steps of reasonable sizes to make good progress towards the optimal solution, this makes training rather hard. An example of a cliff is illustrated in Figure 8.8. The challenges associated with cliffs are discussed in section 4.3.3 of Chapter 4.

There are several solutions to the vanishing and exploding gradient problems, not all of which are equally effective. One solution that is discussed in section 4.5.6 of Chapter 4 is *gradient clipping*. Gradient clipping is well suited to solving the exploding gradient problem. There are two types of clipping that are commonly used. The first is value-based clipping, and the second is norm-based clipping. In value-based clipping, the largest temporal components of the gradient are clipped before adding them. This was the original form of clipping that was proposed by Mikolov in his Ph.D. thesis [336]. The second type of clipping is norm-based clipping. The idea is that when the entire gradient vector has a norm that increases beyond a particular threshold, it is re-scaled back to the threshold. Both types of clipping perform in a similar way, and an analysis is provided in [381].

One observation about suddenly changing curvatures (like cliffs) is that first-order gradients are generally inadequate to fully model local error surfaces. Therefore, a natural solution is to use higher-order gradients. The main challenge with higher-order gradients is that they are computationally expensive. For example, the use of second-order methods

(cf. section 4.6 of Chapter 4) requires the inversion of a Hessian matrix. For a network with 10^6 parameters, this would require the inversion of a $10^6 \times 10^6$ matrix. As a practical matter, this is impossible to do with the computational power available today. However, some clever tricks for implementing second-order methods are discussed in section 4.7 of Chapter 4. These methods have also met with some success in training recurrent neural networks.

The type of instability faced by the optimization process is sensitive to the specific point on the loss surface at which the current solution resides. Therefore, choosing good initialization points is crucial. The work in [147] discusses several types of initialization that can avoid instability in the gradient updates. Using momentum methods (cf. Chapter 4) can also help in addressing some of the instability. A discussion of the power of initialization and momentum in addressing some of these issues is provided in [494]. Often simplified variants of recurrent neural networks, like *echo-state networks*, are used for creating a robust initialization of recurrent neural networks.

Another useful trick that is often used to address the vanishing and exploding gradient problems is that of batch normalization, although the basic approach requires some modifications for recurrent networks [84]. Batch normalization methods are discussed in section 4.8 of Chapter 4. However, a variant known as *layer normalization* is more effective in recurrent networks. Layer normalization methods have been so successful that they have become a standard option while using a recurrent neural network or its variants.

Finally, a number of variants of recurrent neural networks are used to address the vanishing and exploding gradient problems. The first simplification is the use of echo-state networks in which the hidden-to-hidden matrices are randomly chosen, but only the output layers are trained. In the early years, echo-state networks were used as viable alternatives to recurrent neural networks, when it was considered too hard to train recurrent neural networks. However, these methods are too simplified to used in very complex settings. Nevertheless, these methods can still be used for robust initialization in recurrent neural networks [494]. A more effective approach for dealing with the vanishing and exploding gradient problems is to arm the recurrent network with internal memory, which lends more stability to the states of the network. The use of long short-term memory (LSTM) has become an effective way of handing the vanishing and exploding gradient problems. This approach introduces some additional states, which can be interpreted as a kind of long-term memory. The long-term memory provides states that are more stable over time, and also provide a greater level of stability to the gradient-descent process. This approach is discussed in section 8.5.

8.3.1 Layer Normalization

The batch normalization technique discussed in section 4.8 of Chapter 4 is designed to address the vanishing and exploding gradient problems in deep neural networks. In spite of its usefulness in most types of neural networks, the approach faces some challenges in recurrent neural networks. First, the batch statistics vary with the time-layer of the neural network, and therefore different statistics need to be maintained for different time-stamps. Furthermore, the number of layers in a recurrent network is *input-dependent*. Therefore, if a test sequence is longer than any of the training sequences encountered in the data, mini-batch statistics may not be available for some of the time-stamps. In general, the computation of the mini-batch statistics is not equally reliable for different time-layers (irrespective of mini-batch size). Finally, batch normalization is hard to apply to online learning tasks because of its need to collect batch statistics up front. Although batch-

normalization can be adapted to recurrent networks [84], a more effective approach is *layer normalization*.

In layer normalization, the normalization is performed only over a single training instance, although the normalization factor is obtained by using all the current activations *in that layer of only the current instance*. This approach is closer to a conventional neural network operation, and we no longer have the problem of maintaining mini-batch statistics. All the information needed to compute the activations for an instance can be obtained from that instance only!

In order to understand how layer-wise normalization works, we repeat the hidden-to-hidden recursion of page 269:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1})$$

This recursion is prone to unstable behavior because of the multiplicative effect across time-layers. We will show how to modify this recurrence with layer-wise normalization. As in the case of conventional batch normalization of Chapter 4, the normalization is applied to *pre-activation* values before applying the tanh activation function. Therefore, the pre-activation value at the tth time-stamp is computed as follows:

$$\overline{a}_t = W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}$$

Note that $\overline{a}_t$ is a vector with as many components as the number of units in the hidden layer (which we have consistently denoted as p in this chapter). We compute the mean μ_t and standard σ_t of the pre-activation values in $\overline{a}_t$:

$$\mu_t = \frac{\sum_{i=1}^{p} a_{ti}}{p}, \qquad \sigma_t = \sqrt{\frac{\sum_{i=1}^{p} a_{ti}^2}{p} - \mu_t^2}$$

Here, a_{ti} denotes the ith component of the vector $\overline{a}_t$.

As in batch normalization, we have additional learning parameters, associated with each unit. Specifically, for the p units in the tth layer, we have a p-dimensional vector of *gain parameters* $\overline{\gamma}_t$, and a p-dimensional vector of *bias parameters* denoted by $\overline{\beta}_t$. These parameters are analogous to the parameters γ_i and β_i in section 4.8 on batch normalization. The purpose of these parameters is to re-scale the normalized values and add bias in a learnable way. The hidden activations $\overline{h}_t$ of the next layer are therefore computed as follows:

$$\overline{h}_t = \tanh\left(\frac{\overline{\gamma}_t}{\sigma_t} \odot (\overline{a}_t - \overline{\mu}_t) + \overline{\beta}_t\right) \tag{8.7}$$

Here, the notation $\odot$ indicates elementwise multiplication, and the notation $\overline{\mu}_t$ refers to a vector containing p copies of the scalar μ_t. The effect of layer normalization is to ensure that the magnitudes of the activations do not continuously increase or decrease with time-stamp (causing vanishing and exploding gradients), and the learnable parameters add flexibility.

8.4 Echo-State Networks

Echo-state networks represent a simplification of recurrent neural networks. They work well when the dimensionality of the input is small; this is because echo-state networks scale well with the number of temporal units but not with the dimensionality of the input. Therefore, these networks would be a solid option for regression-based modeling of a single or

small number of real-valued time series over a relatively long time horizon. However, they would be a poor choice for modeling text in which the input dimensionality (based on one-hot encoding) would be the size of the lexicon in which the documents are represented. Nevertheless, even in this case, echo-state networks are practically useful in the initialization of weights within the network. Echo-state networks are also referred to as *liquid-state machines* [315], except that the latter uses spiking neurons with binary outputs, whereas echo-state networks use conventional activations like the sigmoid and the tanh functions.

Echo-state networks use *random weights* in the hidden-to-hidden layer and even the input-to-hidden layer, although the dimensionality of the hidden states is almost always much larger than the dimensionality of input states. For a single input series, it is not uncommon to use hidden states of dimensionality about 200. Therefore, only the output layer is trained, which is typically done with a linear layer for real-valued outputs. Note that the training of the output layer simply aggregates the errors at different output nodes, although the weights at different output nodes are still shared. Nevertheless, the objective function would still evaluate to a case of linear regression, which can be trained very simply without the need for backpropagation. Therefore, the training of the echo-state network is very fast.

As in traditional recurrent networks, the hidden-to-hidden layers have nonlinear activations such as the logistic sigmoid function, although tanh activations are also possible. A very important caveat in the initialization of the hidden-to-hidden units is that the largest eigenvector of the weight matrix W_{hh} should be set to 1. This can be easily achieved by first sampling the weights of the matrix W_{hh} randomly from a standard normal distribution, and then dividing each entry by the largest absolute eigenvalue $|\lambda_{max}|$ of this matrix.

$$W_{hh} \Leftarrow W_{hh}/|\lambda_{max}| \tag{8.8}$$

After this normalization, the largest eigenvalue of this matrix will be 1, which corresponds to its *spectral radius*. However, using a spectral radius of 1 can be too conservative because the nonlinear activations will have a dampening effect on the values of the states. For example, when using the sigmoid activation, the *largest* possible partial derivative of the sigmoid is always 0.25, and therefore using a spectral radius much larger than 4 (say, 10) is okay. When using the tanh activation function it would make sense to have a spectral radius of about 2 or 3. These choices would often still lead to a certain level of dampening over time, which is actually a useful regularization because very long-term relationships are generally much weaker than short-term relationships in time-series. One can also tune the spectral radius based on performance by trying different values of the scaling factor γ on held-out data to set $W_{hh} = \gamma W_0$. Here, W_0 is a randomly initialized matrix.

It is recommended to use sparse connectivity in the hidden-to-hidden connections, which is not uncommon in settings involving transformations with random projections. In order to achieve this goal, a number of connections in W_{hh} can be sampled to be non-zero and others are set to 0. This number of connections is typically linear in the number of hidden units. Another key trick is to divide the hidden units into groups indexed $1 \ldots K$ and only allow connectivity between hidden states belonging to with the same index. Such an approach can be shown to be equivalent to training an ensemble of echo-state networks (see. Exercise 2).

Another issue is about setting the input-to-hidden matrices W_{xh}. One needs to be careful about the scaling of this matrix as well, or else the effect of the inputs in each time-stamp can seriously damage the information carried in the hidden states from the previous time-stamp. Therefore, the matrix W_{xh} is first chosen randomly to W_1, and then it is scaled with different values of the hyper-parameter β in order to determine the final matrix $W_{xh} = \beta W_1$ that gives the best accuracy on held-out data.

The core of the echo-state network is based on a very old idea that expanding the number of features of a data set with a nonlinear transformation can often increase the expressive power of the input representation. For example, the RBF network (cf. Chapter 6) and the kernel support-vector machine both gain their power from expansion of the underlying feature space according to Cover's theorem on separability of patterns [87]. The only difference is that the echo-state network performs the feature expansion with random projection; such an approach is not without precedent because various types of random transformations are also used in machine learning as fast alternatives to kernel methods [401, 536]. It is noteworthy that feature expansion is primarily effective through nonlinear transformations, and these are provided through the activations in the hidden layers. In a sense, the echo-state method works using a similar principle to the RBF network in the temporal domain, just as the recurrent neural network is the replacement of feed-forward networks in the temporal domain. Just as the RBF network uses very little training for extracting the hidden features, the echo-state network uses little training for extracting the hidden features and instead relies on the randomized expansion of the feature space.

When used on time-series data, the approach provides excellent results on predicting values far out in the future. The key trick is to choose target output values at a time-stamp t that correspond to the time-series input values at $t+k$, where k is the lookahead required for forecasting. In other words, an echo-state network is an excellent nonlinear autoregressive technique for modeling time-series data. One can even use this approach for forecasting multivariate time-series, although it is inadvisable to use the approach when the number of time series is very large. This is because the dimensionality of hidden states required for modeling would be simply too large. A detailed discussion on the application of the echo-state network for time-series modeling is provided in section 8.7.6. A comparison with respect to traditional time-series forecasting models is also provided in the same section.

Although the approach cannot be realistically used for very high-dimensional inputs (like text), it is still very useful for initialization [494]. The basic idea is to initialize the recurrent network by using its echo-state variant to train the output layer. Furthermore, a proper scaling of the initialized values W_{hh} and W_{xh} can be set by trying different values of the scaling factors β and γ (as discussed above). Subsequently, traditional backpropagation is used to train the recurrent network. This approach can be viewed as a lightweight pretraining for recurrent networks.

A final issue is about the sparsity of the weight connections. Should the matrix W_{hh} be sparse? This is generally a matter of some controversy and disagreement; while sparse connectivity of echo-state networks has been recommended since the early years [230], the reasons for doing so are not very clear. The original work [230] states that sparse connectivity leads to a decoupling of the individual subnetworks, which encourages the development of individual dynamics. This seems to be an argument for increased diversity of the features learned by the echo-state network. If decoupling is indeed the goal, it would make a lot more sense to do so explicitly, and divide the hidden states into disconnected groups. Such an approach has an ensemble-centric interpretation. It is also often recommended to increase sparsity in methods involving random projections for improved efficiency of the computations. Having dense connections can cause the activations of different states to be embedded in the multiplicative noise of a large number of Gaussian random variables, and therefore more difficult to extract.

8.5 Long Short-Term Memory (LSTM)

As discussed in section 8.3, recurrent neural networks have problems associated with vanishing and exploding gradients [215, 381, 382]. This is a common problem in neural network updates where successive multiplication by the matrix $W^{(k)}$ is inherently unstable; it either results in the gradient disappearing during backpropagation, or in it blowing up to large values in an unstable way. This type of instability is the direct result of successive multiplication with the (recurrent) weight matrix at various time-stamps.

A second problem is that a recurrent neural network has difficulty in retaining the information in a long segment of text, because each transition erases some of the information learned in previous steps. For example, consider the following sentence: "*Selecting the left branch at the fork was a mistake, and I should have understood that it made better sense to choose the right one.*" In this case, the word "*right*" towards the end of the sentence relates to direction, and the word "*one*" refers to the branch. However, an RNN may have difficulty in understanding this when it reaches the end of the sentence, because much of the relevant information has already been scrambled by successive transitions. One way of viewing this problem is that a neural network that uses only multiplicative updates is good only at learning over short sequences, and is therefore inherently endowed with good short-term memory but poor long-term memory [215]. To address this problem, a solution is to change the recurrence equation for the hidden vector with the use of the LSTM with the use of long-term memory. The operations of the LSTM are designed to have fine-grained control over the data written into this long-term memory.

As in the previous sections, the notation $\overline{h}_t^{(k)}$ represents the hidden states of the kth layer of a multi-layer LSTM. For notational convenience, we also assume that the input layer $\overline{x}_t$ can be denoted by $\overline{h}_t^{(0)}$ (although this layer is obviously not hidden). As in the case of the recurrent network, the input vector $\overline{x}_t$ is d-dimensional, whereas the hidden states are p-dimensional. The LSTM is an enhancement of the recurrent neural network architecture of Figure 8.6 in which we change the recurrence conditions of how the hidden states $\overline{h}_t^{(k)}$ are propagated. In order to achieve this goal, we have an additional hidden vector of p dimensions, which is denoted by $\overline{c}_t^{(k)}$ and referred to as the *cell state*. One can view the cell state as a kind of long-term memory that retains at least a part of the information in earlier states by using a combination of partial "forgetting" and "increment" operations on the previous cell states. It has been shown in [243] that the nature of the memory in $\overline{c}_t^{(k)}$ is occasionally interpretable when it is applied to text data such as literary pieces. For example, one of the p values in $\overline{c}_t^{(k)}$ might change in sign after an opening quotation and then revert back only when that quotation is closed. The upshot of this phenomenon is that the resulting neural network is able to model long-range dependencies in the language or even a specific pattern (like a quotation) extended over a large number of tokens. This is achieved by using a gentle approach to update these cell states over time, so that there is greater persistence in information storage. Persistence in state values avoids the kind of instability that occurs in the case of the vanishing and exploding gradient problems. One way of understanding this intuitively is that if the states in different temporal layers share a greater level of similarity (through long-term memory), it is harder for the gradients with respect to the incoming weights to be drastically different.

As with the multilayer recurrent network, the update matrix is denoted by $W^{(k)}$ and is used to premultiply the column vector $[\overline{h}_t^{(k-1)}, \overline{h}_{t-1}^{(k)}]^T$. However, this matrix is of size[2]

[2]In the first layer, the matrix $W^{(1)}$ is of size $4p \times (p + d)$ because it is multiplied with a vector of size $(p + d)$.

$4p \times 2p$, and therefore pre-multiplying a vector of size $2p$ with $W^{(k)}$ results in a vector of size $4p$. In this case, the updates use four intermediate, p-dimensional vector variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$ that correspond to the $4p$-dimensional vector. The intermediate variables $\overline{i}$, $\overline{f}$, and $\overline{o}$ are respectively referred to as *input*, *forget*, and *output* variables, because of the roles they play in updating the cell states and hidden states. The determination of the hidden state vector $\overline{h}_t^{(k)}$ and the cell state vector $\overline{c}_t^{(k)}$ uses a multi-step process of first computing these intermediate variables and then computing the hidden variables from these intermediate variables. Note the difference between intermediate variable vector $\overline{c}$ and primary cell state $\overline{c}_t^{(k)}$, which have completely different roles. The updates are as follows:

$$\begin{array}{l}\text{Input Gate:}\\ \text{Forget Gate:}\\ \text{Output Gate:}\\ \text{New C.-State:}\end{array} \begin{bmatrix} \overline{i} \\ \overline{f} \\ \overline{o} \\ \overline{c} \end{bmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix} \textbf{[Setting up intermediates]}$$

$$\overline{c}_t^{(k)} = \overline{f} \odot \overline{c}_{t-1}^{(k)} + \overline{i} \odot \overline{c} \textbf{ [Selectively forget and add to long-term memory]}$$

$$\overline{h}_t^{(k)} = \overline{o} \odot \tanh(\overline{c}_t^{(k)}) \textbf{ [Selectively leak long-term memory to hidden state]}$$

Here, the element-wise product of vectors is denoted by "$\odot$," and the notation "sigm" denotes a sigmoid operation. For the very first layer (i.e., $k = 1$), the notation $\overline{h}_t^{(k-1)}$ in the above equation should be replaced with $\overline{x}_t$ and the matrix $W^{(1)}$ is of size $4p \times (p + d)$. In practical implementations, biases are also used[3] in the above updates, although they are omitted here for simplicity. The aforementioned update seems rather cryptic, and therefore it requires further explanation.

The first step in the above sequence of equations is to set up the intermediate variable vectors $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$, of which the first three should *conceptually* be considered binary values, although they are continuous values in $(0, 1)$. Multiplying a pair of binary values is like using an AND gate on a pair of boolean values. We will henceforth refer to this operation as gating. The vectors $\overline{i}$, $\overline{f}$, and $\overline{o}$ are referred to as input, forget, and output gates. In particular, these vectors are conceptually used as boolean gates for deciding (i) whether to add to a cell-state, (ii) whether to forget a cell state, and (iii) whether to allow leakage into a hidden state from a cell state. The use of the binary abstraction for the input, forget, and output variables helps in understanding the types of decisions being made by the updates. In practice, a continuous value in $(0, 1)$ is contained in these variables, which can enforce the effect of the binary gate in a probabilistic way if the output is seen as a probability. In the neural network setting, it is essential to work with continuous functions in order to ensure the differentiability required for gradient updates. The vector $\overline{c}$ contains the newly proposed contents of the cell state, although the input and forget gates regulate how much it is allowed to change the previous cell state (to retain long-term memory).

The four intermediate variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$, are set up using the weight matrices $W^{(k)}$ for the kth layer in the first equation above. Let us now examine the second equation that updates the cell state with the use of some of these intermediate variables:

$$\overline{c}_t^{(k)} = \underbrace{\overline{f} \odot \overline{c}_{t-1}^{(k)}}_{\text{Reset?}} + \underbrace{\overline{i} \odot \overline{c}}_{\text{Increment?}}$$

[3]The bias associated with the forget gates is particularly important. The bias of the forget gate is generally initialized to values greater than 1 [239] because it seems to avoid the vanishing gradient problem at initialization.

This equation has two parts. The first part uses the p forget bits in $\overline{f}$ to decide which of the p cell states from the previous time-stamp to reset[4] to 0, and it uses the p input bits in $\overline{i}$ to decide whether to add the corresponding components from $\overline{c}$ to each of the cell states. Note that such updates of the cell states are in additive form, which is helpful in avoiding the vanishing gradient problem caused by multiplicative updates. One can view the cell-state vector as a continuously updated long-term memory, where the forget and input bits respectively decide (i) whether to reset the cell states from the previous time-stamp and forget the past, and (ii) whether to increment the cell states from the previous time-stamp to incorporate new information into long-term memory from the current word. The vector $\overline{c}$ contains the p amounts with which to increment the cell states, and these are values in $[-1, +1]$ because they are all outputs of the tanh function.

Finally, the hidden states $\overline{h}_t^{(k)}$ are updated using leakages from the cell state. The hidden state is updated as follows:

$$\overline{h}_t^{(k)} = \underbrace{\overline{o} \odot \tanh(\overline{c}_t^{(k)})}_{\text{Leak } \overline{c}_t^{(k)} \text{ to } \overline{h}_t^{(k)}}$$

Here, we are copying a functional form of each of the p cell states into each of the p hidden states, depending on whether the output gate (defined by $\overline{o}$) is 0 or 1. Of course, in the continuous setting of neural networks, partial gating occurs and only a fraction of the signal is copied from each cell state to the corresponding hidden state. It is noteworthy that the final equation does not always use the tanh activation function. The following alternative update may be used:

$$\overline{h}_t^{(k)} = \overline{o} \odot \overline{c}_t^{(k)}$$

As in the case of all neural networks, the backpropagation algorithm is used for training purposes.

In order to understand why LSTMs provide better gradient flows than vanilla RNNs, let us examine the update for a simple LSTM with a single layer and $p = 1$. In such a case, the cell update can be simplified to the following:

$$c_t = c_{t-1} * f + i * c \tag{8.9}$$

Therefore, the partial derivative c_t with respect to c_{t-1} is f, which means that the backward gradient flows for c_t are multiplied with the value of the forget gate f. Because of elementwise operations, this result generalizes to arbitrary values of the state dimensionality p. The biases of the forget gates are often set to high values initially, so that the gradient flows decay relatively slowly. The forget gate f can also be different at different time-stamps, which reduces the propensity of the vanishing gradient problem. The hidden states can be expressed in terms of the cell states as $h_t = o * \tanh(c_t)$, so that one can compute the partial derivative with respect to h_t with the use of a single tanh derivative. In other words, the long-term cell states function as gradient super-highways, which leak into hidden states.

8.6 Gated Recurrent Units (GRUs)

The Gated Recurrent Unit (GRU) can be viewed as a simplification of the LSTM, which does not use explicit cell states. Another difference is that the LSTM directly controls the

[4]Here, we are treating the forget bits as a vector of binary bits, although it contains continuous values in $(0, 1)$, which can be viewed as probabilities. As discussed earlier, the binary abstraction helps us understand the conceptual nature of the operations.

amount of information changed in the hidden state using separate forget and output gates. On the other hand, a GRU uses a single reset gate to achieve the same goal. However, the basic idea in the GRU is quite similar to that of an LSTM, in terms of how it partially resets the hidden states. As in the previous sections, we assume a multilayer architecture (in addition to time layering); therefore, the notation $\overline{h}_t^{(k)}$ represents the hidden states of the kth layer at time-stamp t for $k \geq 1$. For notational convenience, we also assume that the input layer $\overline{x}_t$ can be denoted by $\overline{h}_t^{(0)}$ (although this layer is obviously not hidden). As in the case of LSTM, we assume that the input vector $\overline{x}_t$ is d-dimensional, whereas the hidden states are p-dimensional. The sizes of the transformation matrices in the first layer are accordingly adjusted to account for this fact.

In the case of the GRU, we use two matrices $W^{(k)}$ and $V^{(k)}$ of sizes[5] $2p \times 2p$ and $p \times 2p$, respectively. Pre-multiplying a vector of size $2p$ with $W^{(k)}$ results in a vector of size $2p$, which will be passed through the sigmoid activation to create two intermediate, p-dimensional vector variables $\overline{z}_t$ and $\overline{r}_t$, respectively. The intermediate variables $\overline{z}_t$ and $\overline{r}_t$ are respectively referred to as update and reset gates. The determination of the hidden state vector $\overline{h}_t^{(k)}$ uses a two-step process of first computing these gates, then using them to decide how much to change the hidden vector with the weight matrix $V^{(k)}$:

$$\begin{array}{l}\text{Update Gate:}\\ \text{Reset Gate:}\end{array} \left[\begin{array}{c}\overline{z}\\ \overline{r}\end{array}\right] = \left(\begin{array}{c}\text{sigm}\\ \text{sigm}\end{array}\right) W^{(k)} \left[\begin{array}{c}\overline{h}_t^{(k-1)}\\ \overline{h}_{t-1}^{(k)}\end{array}\right] \quad \textbf{[Set up gates]}$$

$$\overline{h}_t^{(k)} = \overline{z} \odot \overline{h}_{t-1}^{(k)} + (1 - \overline{z}) \odot \tanh\, V^{(k)} \left[\begin{array}{c}\overline{h}_t^{(k-1)}\\ \overline{r} \odot \overline{h}_{t-1}^{(k)}\end{array}\right] \quad \textbf{[Update hidden state]}$$

Here, the element-wise product of vectors is denoted by "$\odot$," and the notation "sigm" denotes a sigmoid operation. For the very first layer (i.e., $k = 1$), the notation $\overline{h}_t^{(k-1)}$ in the above equation should be replaced with $\overline{x}_t$. Furthermore, the matrices $W^{(1)}$ and $V^{(1)}$ are of sizes $2p \times (p + d)$ and $p \times (p + d)$, respectively. One can write the entirety of the function represented in the equations above by a single GRU function:

$$\overline{h}_t^{(k)} = \text{GRU}\left(\overline{h}_t^{(k-1)}, \overline{h}_{t-1}^{(k)}, W^{(k)}, V^{(k)}\right) \tag{8.10}$$

Here, $W^{(k)}$ and $V^{(k)}$ are the matrices used in the kth layer for setting up the gates and updating the hidden states respectively. We have also omitted the mention of biases here, but they are usually included in practical implementations. In the following, we provide a further explanation of these updates and contrast them with those of the LSTM. Just as the LSTM uses input, output, and forget gates to decide how much of the information from the previous time-stamp to carry over to the next step, the GRU uses the update and the reset gates. The GRU does not have a separate internal memory and also requires fewer gates to perform the update from one hidden state to another. Therefore, a natural question arises about the precise role of the update and reset gates. The reset gate $\overline{r}$ decides how much of the hidden state to carry over from the previous time-stamp for a matrix-based update (like a recurrent neural network). The update gate $\overline{z}$ decides the *relative* strength of the contributions of this matrix-based update and a more direct contribution

[5] In the first layer ($k = 1$), these matrices are of sizes $2p \times (p + d)$ and $p \times (p + d)$.

from the hidden vector $\overline{h}_{t-1}^{(k)}$ at the previous time-stamp. By allowing a direct (partial) copy of the hidden states from the previous layer, the gradient flow becomes more stable during backpropagation. The update gate of the GRU simultaneously performs the role of the input and forget gates in the LSTM in the form of $\overline{z}$ and $1 - \overline{z}$, respectively. However, the mapping between the GRU and the LSTM is not precise, because it performs these updates directly on the hidden state (and there is no cell state). Although the GRU is a closely related simplification of the LSTM, it should not be seen as a special case of the LSTM.

In order to understand why GRUs provide better performance than vanilla RNNs, let us examine a GRU with a single layer and single state dimensionality $p = 1$. In such a case, the update equation of the GRU can be written as follows:

$$h_t = z \cdot h_{t-1} + (1 - z) \cdot \tanh[v_1 \cdot x_t + v_2 \cdot r \cdot h_{t-1}] \tag{8.11}$$

Note that layer superscripts are missing in this single-layer case. Here, v_1 and v_2 are the two elements of the 2×1 matrix V. Then, it is easy to see the following:

$$\frac{\partial h_t}{\partial h_{t-1}} = z + (\text{Additive Terms}) \tag{8.12}$$

Backward gradient flow is multiplied with this factor. Here, the term $z \in (0, 1)$ helps in passing *unimpeded* gradient flow and makes computations more stable. Furthermore, since the additive terms heavily depend on $(1 - z)$, the overall multiplicative factor tends to be closer to 1 even when z is small.

A comparison of the LSTM and the GRU is provided in [74, 239]. The two models are shown to be roughly similar in performance, and the relative performance seems to depend on the task at hand. The GRU is simpler and enjoys the advantage of greater ease of implementation and efficiency. It might generalize slightly better with less data because of a smaller parameter footprint [74], although the LSTM would be preferable with an increased amount of data. The work in [239] also discusses several practical implementation issues associated with the LSTM. The LSTM has been more extensively tested than the GRU, simply because it is an older architecture and enjoys widespread popularity. As a result, it is generally seen as a safer option, particularly when working with longer sequences and larger data sets. The work in [167] also showed that none of the variants of the LSTM can reliably outperform it in a consistent way. This is because of the explicit internal memory and the greater gate-centric control in updating the LSTM.

8.7 Applications of Recurrent Neural Networks

Recurrent neural networks have numerous applications in machine learning applications, which are associated with information retrieval, speech recognition, and handwriting recognition. Text data forms the predominant setting for applications of RNNs, although there are several applications to computational biology as well. Most of the applications of RNNs fall into one of two categories:

1. *Conditional language modeling:* When the output of a recurrent network is a language model, one can enhance it with context in order to provide a relevant output to the context. In most of these cases, the context is the neural output of another neural network. To provide one example, in image captioning the context is the neural representation of an image provided by a convolutional network, and the language model

provides a caption for the image. In machine translation, the context is the representation of a sentence in a source language (produced by another RNN), and the language model in the target language provides a translation.

2. *Leveraging token-specific outputs:* The outputs at the different tokens can be used to learn other properties than a language model. For example, the labels output at different time-stamps might correspond to the properties of the tokens (such as their parts of speech). In handwriting recognition, the labels might correspond to the characters. In some cases, all the time-stamps might not have an output, but the end-of-sentence marker might output a label for the entire sentence. This approach is referred to as sentence-level classification, and is often used in sentiment analysis. In some of these applications, bidirectional recurrent networks are used because the context on both sides of a word is helpful.

The following material will provide an overview of the numerous applications of recurrent neural networks. In most of these cases, we will use a single-layer recurrent network for ease in explanation and pictorial illustration. However, in most cases, a multi-layer LSTM is used. In other cases, a bidirectional LSTM is used, because it provides better performance. Replacing a single-layer RNN with a multi-layer/bidirectional LSTM in any of the following applications is straightforward. Out broader goal is to illustrate how this *family* of architectures can be used in these settings.

8.7.1 Contextualized Word Embeddings with ELMo

Recurrent neural networks are extremely powerful at constructing pre-trained language models, which are then used for a variety of downstream applications. An important observation about language models is that *they are unsupervised multitask learners*. For example, most of the applications discussed in this section can be accomplished more efficiently, if one starts with a language model, and then performs task-specific fine-tuning. The amount of effort involved in performing the fine tuning is typically much smaller than what is required than training the model from scratch. This is essentially an example of *transfer learning*, wherein the training work that is done up front is reused for task-specific training. The main caveat for building a useful pre-trained language model is that it needs to be large and built on a sufficient training data. This type of model is also useful in building *contextualized* word representations that have an advantage over models like*word2vec* in terms of creating *sentence-specific* embeddings. For example, consider the following pair of sentences:

He used a stick to pry open the gate.
The postage stamp did not stick properly to the envelope.

The *word2vec* embedding is not sentence-specific, and it will therefore give the same representation to the word "stick" in both cases, which can be viewed as a weighted average of the embeddings of the two contexts, based on relative presence in the *training* corpus. However, in some applications, one might want to derive a word embedding that is *specific to the particular sentence at hand*. This type of approach is useful in many applications like *word sense disambiguation* in which we wish to determine the correct sense in which a particular word is used.

The ELMo language model [386] is constructed by using a bidirectional recurrent neural network (cf. section 8.2.3), and more specifically, a bidirectional LSTM. After it has been trained, contextualized embeddings for the words in a given sentence can be generated

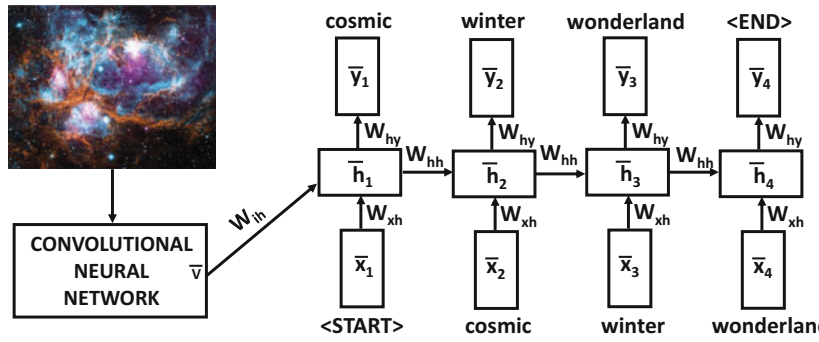


Figure 8.9: Example of image captioning with a recurrent neural network. An additional convolutional neural network is required for representational learning of the images. The image is represented by the vector $\overline{v}$, which is the output of the convolutional neural network. The inset image is by courtesy of the National Aeronautics and Space Administration (NASA).

by feeding the sentence to the trained network. The contextualized representation of a word is given by the concatenation of the forward and backward (hidden) states at that position of the sentence. If a multilayer recurrent neural network is used, then the forward and backward states of all the layers in a particular position are concatenated. This type of word representation can be very useful for token-wise applications like identifying the part of speech in a particular sentence. Furthermore, the forward hidden state at the end of the sentence and the backward hidden state at the beginning of the sentence can be concatenated in order to obtain a representation of the full sentence. *Language models can also be used for multitask training, because the training sequences can start with a task-specific prefix (e.g., "Translate from English to German:"), which is followed by the source and target sequences separated by special tokens.* The pretrained language model of ELMo has been used for many natural language processing applications, such as *entity extraction*, *word-sense disambiguation*, and *sentiment analysis*.

8.7.2 Application to Automatic Image Captioning

In image captioning, the training data consists of image-caption pairs. For example, the image[6] in the left-hand side of Figure 8.9 is obtained from the National Aeronautics and Space Administration Web site. This image is captioned "*cosmic winter wonderland.*" One might have hundreds of thousands of such image-caption pairs. These pairs are used to train the weights in the neural network. Once the training has been completed, the captions are predicted for unknown test instances. Therefore, one can view this approach as an instance of image-to-sequence learning.

One issue in the automatic captioning of images is that a separate neural network is required to learn the representation of the images. A common architecture to learn the representation of images is the *convolutional neural network*. A detailed discussion of convolutional neural networks is provide in Chapter 9. Consider a setting in which the convolutional neural network produces the q-dimensional vector $\overline{v}$ as the output representation.

[6] https://www.nasa.gov/mission_pages/chandra/cosmic-winter-wonderland.html

This vector is then used as an input to the neural network, but only[7] at the first time-stamp. To account for this additional input, we need another $p \times q$ matrix W_{ih}, which maps the image representation to the hidden layer. Therefore, the update equations for the various layers now need to be modified as follows:

$$\overline{h}_1 = \tanh(W_{xh}\overline{x}_1 + W_{ih}\overline{v})$$
$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}) \quad \forall t \geq 2$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

An important point here is that the convolutional neural network and the recurrent neural network are not trained in isolation. Although one might train them in isolation in order to create an initialization, the final weights are always trained jointly by running each image through the network and matching up the predicted caption with the true caption. In other words, for each image-caption pair, the weights in both networks are updated when errors are made in predicting any particular token of the caption. In practice, the errors are soft because the tokens at each point are predicted probabilistically. Such an approach ensures that the learned representation $\overline{v}$ of the images is sensitive to the specific application of predicting captions.

After all the weights have been trained, a test image is input to the entire system and passed through both the convolutional and recurrent neural network. For the recurrent network, the input at the first time-stamp is the <START> token and the representation of the image. At later time-stamps, the input is the most likely token predicted at the previous time-stamp. One can also use beam search to keep track of the b most likely sequence prefixes to expand on at each point. This approach is not very different from the language generation approach discussed in section 8.2.1, except that it is conditioned on the image representation that is input to the model in the first time-stamp of the recurrent network. This results in the prediction of a relevant caption for the image.

8.7.3 Sequence-to-Sequence Learning and Machine Translation

Just as one can put together a convolutional neural network and a recurrent neural network to perform image captioning, one can put together two recurrent networks to translate one language into another. Such methods are also referred to as *sequence-to-sequence* learning because a sequence in one language is mapped to a sequence in another language. In principle, sequence-to-sequence learning can have wide-spectrum applicability by framing any input-output pair (e.g., classification or question answering) as a text-to-text translation.

In the following, we provide a simple solution to machine translation with recurrent neural networks, although such applications are rarely addressed directly with the simple forms of recurrent neural networks. Rather, a variation of the recurrent neural network, such as the long short-term memory (LSTM) or Gated Recurrent Unit (GRU) model is used. More recently, *transformers* (cf. section 12.2.5 of Chapter 12) have been used widely.

In the machine translation application, two different RNNs are hooked end-to-end, just as a convolutional neural network and a recurrent neural network are hooked together for image captioning. The first recurrent network uses the words from the source language as input. No outputs are produced at these time-stamps and the successive time-stamps accumulate knowledge about the source sentence in the hidden state. Subsequently, the end-of-sentence symbol is encountered, and the second recurrent network starts by outputting

[7] In principle, one can also allow it to be input at all time-stamps, but it only seems to worsen performance.

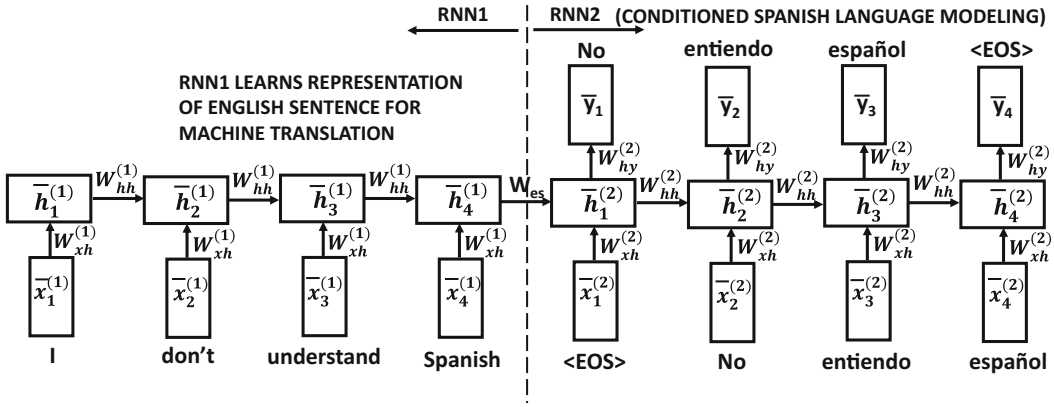


Figure 8.10: Machine translation with recurrent neural networks. Note that there are two separate recurrent networks with their own sets of shared weights. The output of $\overline{h}_4^{(1)}$ is a fixed length encoding of the 4-word English sentence.

the first word of the target language. The next set of states in the second recurrent network output the words of the sentence in the target language one by one. These states also use the words of the target language as input, which is available for the case of the training instances but not for test instances (where predicted values are used instead). This architecture is shown in Figure 8.10.

The architecture of Figure 8.10 is similar to that of an autoencoder, and can even be used with pairs of identical sentences in the same language to create fixed-length representations of sentences. The two recurrent networks are denoted by RNN1 and RNN2, and their weights are not the same. For example, the weight matrix between two hidden nodes at successive time-stamps in RNN1 is denoted by $W_{hh}^{(1)}$, whereas the corresponding weight matrix in RNN2 is denoted by $W_{hh}^{(2)}$. The weight matrix W_{es} of the link joining the two neural networks is special, and can be independent of either of the two networks. This is necessary if the sizes of the hidden vectors in the two RNNs are different because the dimensions of the matrix W_{es} will be different from those of both $W_{hh}^{(1)}$ and $W_{hh}^{(2)}$. As a simplification, one can use[8] the same size of the hidden vector in both networks, and set $W_{es} = W_{hh}^{(1)}$. The weights in RNN1 are devoted to learning an encoding of the input in the source language, and the weights in RNN2 are devoted to using this encoding in order to create an output sentence in the target language. One can view this architecture in a similar way to the image captioning application, except that we are using two recurrent networks instead of a convolutional-recurrent pair. The output of the final hidden node of RNN1 is a fixed-length encoding of the source sentence. Therefore, irrespective of the length of the sentence, the encoding of the source sentence depends on the dimensionality of the hidden representation.

The grammar and length of the sentence in the source and target languages may not be the same. In order to provide a grammatically correct output in the target language, RNN2 needs to learn its language model. It is noteworthy that the units in RNN2 associated with the target language have both inputs and outputs arranged in the same way as a language-modeling RNN. At the same time, the output of RNN2 is conditioned on the input it

[8]The original work in [494] seems to use this option. In the Google Neural Machine Translation system [603], this weight is removed. This system is now used in Google Translate.

receives from RNN1, which effectively causes language translation. In order to achieve this goal, training pairs in the source and target languages are used. The approach passes the source-target pairs through the architecture of Figure 8.10 and learns the model parameters with the use of the backpropagation algorithm. Since only the nodes in RNN2 have outputs, only the errors made in predicting the target language words are backpropagated to train the weights in both neural networks. The two networks are jointly trained, and therefore the weights in both networks are optimized to the errors in the translated outputs of RNN2. As a practical matter, this means that the internal representation of the source language learned by RNN1 is highly optimized to the machine translation application, and is very different from one that would be learned if one had used RNN1 to perform language modeling of the source sentence. After the parameters have been learned, a sentence in the source language is translated by first running it through RNN1 to provide the necessary input to RNN2. Aside from this contextual input, another input to the first unit of RNN2 is the <EOS> tag, which causes RNN2 to output the likelihoods of the first token in the target language. The most likely token using beam search (cf. section 8.2.1) is selected and used as the input to the recurrent network unit in the next time-stamp. This process is recursively applied until the output of a unit in RNN2 is also <EOS>. As in section 8.2.1, we are generating a sentence from the target language using a language-modeling approach, except that the specific output is conditioned on the internal representation of the source sentence.

The use of neural networks for machine translation is relatively recent. Recurrent neural network models have a sophistication that greatly exceeds that of traditional machine translation models. The latter class of methods uses phrase-centric machine learning, which is often not sophisticated enough to learn the subtle differences between the grammars of the two languages. In practice, deep models with multiple layers are used to improve the performance.

One weakness of such translation models is that they tend to work poorly when the sentences are long. Numerous solutions have been proposed to solve the problem. A recent solution is that the sentence in the source language is input in the *opposite order* [494]. This approach brings the first few words of the sentences in the two languages closer in terms of their time-stamps within the recurrent neural network architecture. As a result, the first few words in the target language are more likely to be predicted correctly. The correctness in predicting the first few words is also helpful in predicting the subsequent words, which are also dependent on a neural language model in the target language.

Other Natural Language Processing Applications

Sequence-to-sequence learning has numerous applications beyond machine translation. Some examples are as follows:

- In *abstractive text summarization*, the source and target sequences correspond to the original paragraph and the summarized paragraph. It is assumed that some examples of such summaries are available.
- In *question-answering over knowledge graphs*, the question is often posed in natural language, which then needs to be translated in a *structured query language* like SPARQL. In such cases, the training pairs will correspond to the informal and formal representations of questions. For example, one might have training pairs of questions as follows:

$$\underbrace{\text{What is the capital of China? } < \text{EOQ1} >}_{\text{Natural language question}} \quad \underbrace{\textbf{CapitalOf}(\textit{China}, ?) < \text{EOQ2} >}_{\text{Formal Representation}}$$

The expression on the right-hand side is a structured question, which queries for entities of different types such as persons, places, and organizations. In practice, the structured queries will be explicitly represented using SPARQL (although we have written it in the form of a relation). The first step would be to convert the question into an internal representation like the one above, which is more prone to query answering. This conversion can be done using training pairs of questions and their internal representations in conjunction with an recurrent network.

Sequence-to-sequence models can also be used to construct various types of text autoencoders for unsupervised learning. In fact, most tasks with an input-output structure can be modeled as text-to-text tasks by treating the input and the output as two text sequences (e.g., input sentence followed by output text label for sentence classification). This is the general principle behind the design of some pretraining approaches such as T5 [400], although this method uses *transformers* (cf. Chapter 12) rather than recurrent neural networks.

8.7.4 Application to Sentence-Level Classification

In this problem, each sentence is treated as a training (or test) instance for classification purposes. Sentence-level classification is generally a more difficult problem than document-level classification because sentences are short, and there is often not enough evidence in the vector space representation to perform the classification accurately. However, the sequence-centric view is more powerful and can often be used to perform more accurate classification. The RNN architecture for sentence-level classification is shown in Figure 8.11. Note that the only difference from Figure 8.12 is that we no longer care about the outputs at each node but defer the class output to the end of the sentence. In other words, a single class label is predicted at the very last time-stamp of the sentence, and it is used to backpropagate the class prediction errors.

Sentence-level classification is often leveraged in *sentiment analysis*. This problem attempts to discover how positive or negative users are about specific topics by analyzing the content of a sentence [7]. For example, one can use sentence-level classification to determine whether or not a sentence expresses a positive sentiment by treating the sentiment polarity as the class label. In the example shown in Figure 8.11, the sentence clearly indicates a

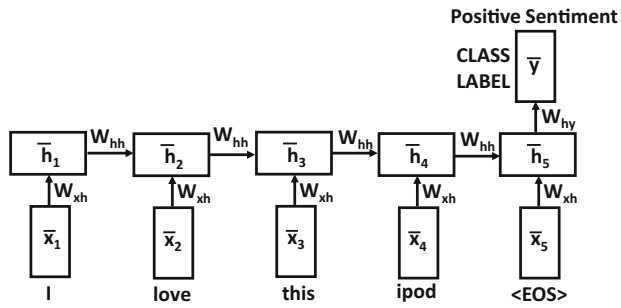


Figure 8.11: Example of sentence-level classification in a sentiment analysis application with the two classes "*positive sentiment*" and "*negative sentiment.*"

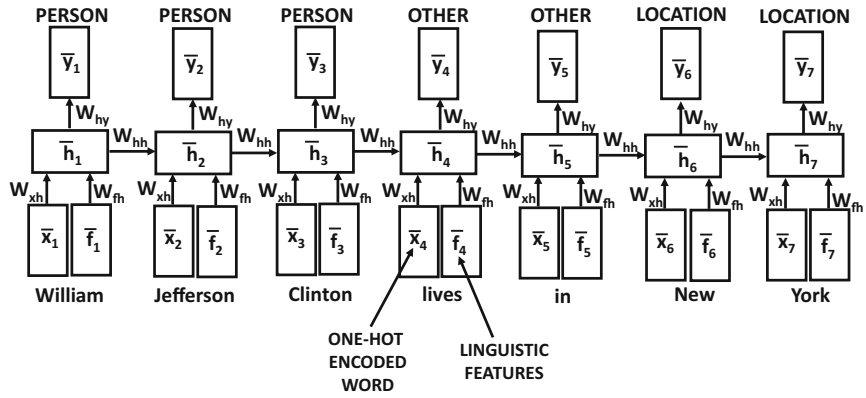


Figure 8.12: Token-wise classification with linguistic features

positive sentiment. Note, however, that one cannot simply use a vector space representation containing the word "*love*" to infer the positive sentiment. For example, if words such as "*don't*" or "*hardly*" occur before "*love*", the sentiment would change from positive to negative. Such words are referred to as *contextual valence shifters* [391], and their effect can be modeled only in a sequence-centric setting. Recurrent neural networks can handle such settings because they use the accumulated evidence over the specific sequence of words in order to predict the class label. One can also combine this approach with linguistic features. In the next section, we show how to use linguistic features for token-level classification; similar ideas also apply to the case of sentence-level classification.

8.7.5 Token-Level Classification with Linguistic Features

The numerous applications of token-level classification include information extraction and text segmentation. In information extraction, specific words or combinations of words are identified that correspond to persons, places, or organizations. The linguistic features of the word (capitalization, part-of-speech, orthography) are more important in these applications than in typical language modeling or machine translation applications. Nevertheless, the methods discussed in this section for incorporating linguistic features can be used for any of the applications discussed in earlier sections. For the purpose of discussion, consider a *named-entity recognition application* in which every entity is to be classified as one of the categories corresponding to person (P), location (L), and other (O). In such cases, each token in the training data has one of these labels. An example of a possible training sentence is as follows:

$$\underbrace{\text{William}}_{P}\ \underbrace{\text{Jefferson}}_{P}\ \underbrace{\text{Clinton}}_{P}\ \underbrace{\text{lives}}_{O}\ \underbrace{\text{in}}_{O}\ \underbrace{\text{New}}_{L}\ \underbrace{\text{York}}_{L}.$$

In practice, the tagging scheme is often more complex because it encodes information about the beginning and end of a set of contiguous tokens with the same label. For test instances, the tagging information about the tokens is not available.

The recurrent neural network can be defined in a similar way as in the case of language modeling applications, except that the outputs are defined by the tags rather than the next set of words. The input at each time-stamp t is the one-hot encoding $\overline{x}_t$ of the token, and the output $\overline{y}_t$ is the tag. Furthermore, we have an additional set of q-dimensional linguistic features $\overline{f}_t$ associated with the tokens at time-stamp t. These linguistic features

might encode information about the capitalization, orthography, capitalization, and so on. The hidden layer, therefore, receives two separate inputs from the tokens and from the linguistic features. The corresponding architecture is illustrated in Figure 8.12. We have an additional $p \times q$ matrix W_{fh} that maps the features $\overline{f}_t$ to the hidden layer. Then, the recurrence condition at each time-stamp t is as follows:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{fh}\overline{f}_t + W_{hh}\overline{h}_{t-1})$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

The main innovation here is in the use of an additional weight matrix for the linguistic features. The change in the type of output tag does not affect the overall model significantly. In some variations, it might also be helpful to *concatenate* the linguistic and token-wise features into as separate *embedding layer*, rather than adding them. The work in [590] provides an example in the case of recommender systems, although the principle can also be applied here. The overall learning process is also not significantly different. In token-level classification applications, it is sometimes helpful to use bidirectional recurrent networks in which recurrence occurs in both temporal directions [454].

8.7.6 Time-Series Forecasting and Prediction

Recurrent neural networks present a natural choice for time-series forecasting and prediction. The main difference from text is that the input units are real-valued vectors rather than (discrete) one-hot encoded vectors. For real-valued prediction, the output layer always uses linear activations, rather than the softmax function. In the event that the output is a discrete value (e.g., identifier of a specific event), it is also possible to use discrete outputs with softmax activation. Although any of the variants of the recurrent neural network (e.g., LSTM or GRU) can be used, one of the common problems in time-series analysis is that such sequences can be extremely long. Even though the LSTM and the GRU provide a certain level of protection with increased time-series length, there are limitations to the performance. This is because LSTMs and GRUs do degrade for series beyond certain lengths. Many time-series can have a very large number of time-stamps with various types of short- and long-term dependencies. The prediction and forecasting problems present unique challenges in these cases.

However, a number of useful solutions exist, at least in cases where the number of time-series to be forecasted is not too large. The most effective method is the use of the echo-state network (cf. section 8.4), in which it is possible to effectively forecast and predict both real-valued and discrete observations with a *small* number of time-series. The caveat that the number of inputs is small is an important one, because echo-state networks rely on randomized expansion of the feature space via the hidden units (see section 8.4). If the number of original time series is too large, then it may not turn out to be practical to expand the dimensionality of the hidden space sufficiently to capture this type of feature engineering. It is noteworthy that the vast majority of forecasting models in the time-series literature are, in fact, univariate models. A classical example is the *autoregressive model* (AR), which uses the immediate window of history in order to perform forecasting.

The use of an echo-state network in order to perform time-series regression and forecasting is straightforward. At each time-stamp, the input is a vector of d values corresponding to the d different time series that are being modeled. It is assumed that the d time series are synchronized, and this is often accomplished by preprocessing and interpolation. The output at each time-stamp is the predicted value. In forecasting, the predicted value is simply the value(s) of the different time-series at k units ahead. One can view this approach

as the time-series analog of language models with discrete sequences. It is also possible to choose an output corresponding to a time-series not present in the data (e.g., predicting one stock price from another) or to choose an output corresponding to a discrete event (e.g., equipment failure). The main differences among all these cases lie in the specific choice of the loss function for the output at hand. In the specific case of time-series forecasting, a neat relationship can be shown between autoregressive models and echo-state networks.

Relationship with Autoregressive Models

An *autoregressive model* models the values of a time-series as a linear function of its immediate history of length p. The p coefficients of this model are learned with linear regression. Echo-state networks can be shown to be closely related to autoregressive models, in which the connections of the hidden-to-hidden matrix are sampled in a particular way. The additional power of the echo-state network over an autoregressive model arises from the nonlinearity used in the hidden-to-hidden layer. In order to understand this point, we will consider the special case of an echo-state network in which its input corresponds to a single time series and the hidden-to-hidden layers have linear activations. Now imagine that we could somehow choose the hidden-to-hidden connections in such a way that the values of the hidden state in each time-stamp is exactly equal to the values of the time-series in the last p ticks. What kind of sampled weight matrix would achieve this goal?

First, the hidden state needs to have p units, and therefore the size of W_{hh} is $p \times p$. It is easy to show that a weight matrix W_{hh} that shifts the hidden state by one unit and copies the input value to the vacated state caused by the shifting will result in a hidden state, which is exactly the same as the last window of p points. In other words, the matrix W_{hh} will have exactly $(p-1)$ non-zero entries of the form $(i, i+1)$ for each $i \in \{1 \ldots p-1\}$. As a result, pre-multiplying any p-dimensional column vector $\overline{h}_t$ with W_{hh} will shift the entries of $\overline{h}_t$ by one unit. For a 1-dimensional time-series, the element x_t is a 1-dimensional input into the tth hidden state of the echo state network, and W_{xh} is therefore of size $p \times 1$. Setting only the entry $(p, 0)$ of W_{xh} to 1 and all other entries to 0 will result in copying x_t into the first element of $\overline{h}_t$. The matrix W_{hy} is a $1 \times p$ matrix of *learned weights*, so that $W_{hy}\overline{h}_t$ yields the prediction $\hat{y}_t$ of the observed value y_t. In autoregressive modeling, the value of y_t is simply set to x_{t+k} for some lookahead k, and the value of k is often set to 1. It is noteworthy that the matrices W_{hh} and W_{xh} are fixed, and only W_{hy} needs to learned. This process leads to the development of a model that is identical to the time-series autoregressive model [3].

The main difference of the time-series autoregressive model from the echo-state network is that the latter fixes W_{hh} and W_{xh} randomly, and uses much larger dimensionalities of the hidden states. Furthermore, nonlinear activations are used in the hidden units. As long as the spectral radius of W_{hh} is (slightly) less than 1, a random choice of the matrices W_{hh} and W_{xh} with linear activations can be viewed as a decay-based variant of the autoregressive model. This is because the matrix W_{hh} only performs a random (but slightly decaying) transformation of the previous hidden state. Using a decaying random projection of the previous hidden state intuitively achieves similar goals as a sliding window-shifted copy of the previous state. The precise spectral radius of W_{hh} governs the rate of decay. With a sufficient number of hidden states, the matrix W_{hy} provides enough degrees of freedom to model any decay-based function of recent history. Furthermore, the proper scaling of the W_{xh} ensures that the most recent entry is not given too much or too little weight. Note that echo-state networks do test different scalings of the matrix W_{xh} to ensure that the effect of this input does not wipe out the contributions from the hidden states. The nonlinear

activations in the echo-state network give greater power to this approach over a time-series autoregressive model. In a sense, echo-state networks can model complex nonlinear dynamics of the time-series, unlike an off-the-shelf autoregressive model.

8.7.7 Temporal Recommender Systems

Several solutions [481, 554, 590] have been proposed in recent years for temporal modeling of recommender systems. Some of these methods use temporal aspects of users, whereas others use temporal aspects of users and items. One observation is that the properties of items tend to be more strongly fixed in time than the properties of users. Therefore, solutions that use the temporal modeling only at the user level are often sufficient. However, some methods [554] perform the temporal modeling both at the user level and at the item level.

In the following, we discuss a simplification of the model discussed in [481]. In temporal recommender systems, the time-stamps associated with user ratings are leveraged for the recommendation process. Consider a case in which the observed rating of user i for item j at time-stamp t is denoted by r_{ijt}. For simplicity, we assume that the time-stamp t is simply the index of the rating in the sequential order it was received (although many models use the wall-clock time). Therefore, the sequence being modeled by the RNN is a sequence of rating values associated with the content-centric representations of the users and items to which the rating belongs. Therefore, we want to model the value of the rating as a function of content-centric inputs at each time-stamp.

We describe these content-centric representations below. The prediction of the rating r_{ijt} is assumed to be depend on (i) static features associated with the item, (ii) static features associated with the user, and (iii) the dynamic features associated with the user. The static features associated with the item might be item titles or descriptions, and one can create a bag-of-words representation of the item. The static features associated with the user might be a user-specific profile or a fixed history of accesses of this user, which does not change over the data set. The static features associated with the users are also typically represented as a bag of words, and one can even consider item-rating pairs as pseudo-keywords in order to combine user-specified keywords with ratings activity. In the case where ratings activity is used, a fixed history of accesses of the user is always leveraged for designing static features. The dynamic user features are more interesting because they are based on the dynamically changing user access history. In this case, a short history of item-rating pairs can be used as pseudo-keywords, and a bag-of-words representation can be created at time-stamp t.

In several cases, explicit ratings are not available, but implicit feedback data is available corresponding to a user clicking on an item. In the event that implicit feedback is used, negative sampling becomes necessary in which user-item pairs for which activity has not occurred are included in the sequence at random. This approach can be viewed as a hybrid between a content-based and collaborative recommendation approach. While it does use the user-item-rating triplets like a traditional recommender model, the content-centric representations of the users and items are input at each time-stamp. However, the inputs at different time-stamps correspond to different user-item pairs, and therefore the collaborative power of the patterns of ratings among different users and items is used as well.

The overall architecture of this recommender system is illustrated in Figure 8.13. It is evident that this architecture contains three different subnetworks to create feature embeddings out of static item features, static user features, and dynamic user features. The first two of these three are feed-forward networks, whereas the last of them is a recurrent neural network. First, the embeddings from the two user-centric networks are fused using either concatenation or element-wise multiplication. In the latter case, it is necessary to create

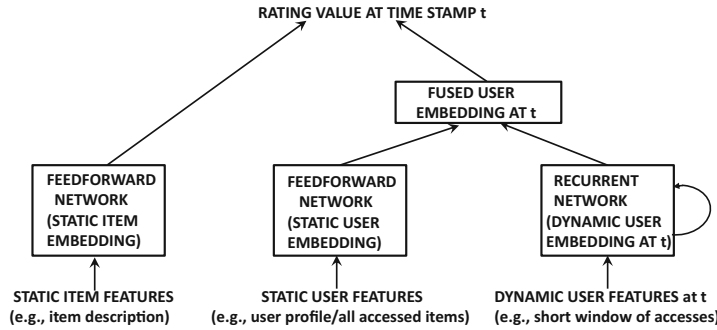


Figure 8.13: Recommendations with recurrent neural networks. At each time-stamp, the static/dynamic user features and static item features are input, and a rating value is output for that user-item combination.

embeddings of the same dimensionality for static and dynamic user features. Then, this fused user embedding at time-stamp t and the static item embedding is used to predict the rating at time-stamp t. For implicit feedback data, one can predict probabilities of positive activity for a particular user-item pair. The chosen loss function depends on the nature of the rating being predicted. The training algorithm needs to work with a consecutive sequence of training triplets (of some fixed mini-batch size) and backpropagate to the static and dynamic portions of the network simultaneously.

The aforementioned presentation has simplified several aspects of the training procedure presented in [481]. For example, it is assumed that a single rating is received at each time-stamp t, and that a fixed time-horizon is sufficient for temporal modeling. In reality, different settings might required different levels of granularity at which temporal aspects are handled. Therefore, the work in [481] proposes methods to address varying levels of granularity in the modeling process. It is also possible to perform the recommendation under a pure collaborative filtering regime without using content-centric features in any way. For example, it is possible[9] to adapt the recommender system discussed in section 3.5 of Chapter 3 by using a recurrent neural network (cf. Exercise 3).

Another recent work [590] treats the problem as that of working with product-action-time triplets at an e-commerce site. The idea is that a site logs sequential actions performed by each user to various products, such as visiting a product page from a homepage, category page, or sales page, and that of actually buying the product. Each action has a *dwell time*, which indicates the amount of time that the user spends in performing that action. The dwell time is discretized into a set of intervals, which would be uniform or geometric, depending on the application at hand. It makes sense to discretize the time into geometrically increasing intervals.

One sequence is collected for each user, corresponding to the actions performed by the user. One can represent the rth element of the sequence as $(\overline{p}_r, \overline{a}_r, \overline{t}_r)$, where $\overline{p}_r$ is the one-hot encoded product, $\overline{a}_r$ is the one-hot encoded action, and $\overline{t}_r$ is the one-hot encoded discretized value of the time interval. Each of $\overline{p}_r$, $\overline{a}_r$, and $\overline{t}_r$ is a one-hot encoded vector. An embedding layer with weight matrices W_p, W_a, and W_t is used to create the representation $\overline{e}_r = (W_p\overline{p}_r, W_a\overline{a}_r, W_t\overline{t}_r)$. These matrices were pretrained with *word2vec* training applied

[9]Even though the adaptation from section 3.5 is the most natural and obvious one, we have not seen it elsewhere in the literature. Therefore, it might be an interesting exercise for the reader to implement the adaptation of Exercise 3.

to sequences extracted from the e-commerce site. Subsequently, the input to the recurrent neural network is $\overline{e}_1 \ldots \overline{e}_T$, which was used to predict the outputs $\overline{o}_1 \ldots \overline{o}_T$. The output at the time-stamp t corresponds to the next action of the user at that time-stamp. Note that the embedding layer is also attached to the recurrent network, and it is fine-tuned during backpropagation (beyond its *word2vec* initialization). The original work [590] also adds an attention layer, although good results can be obtained even without this layer.

8.7.8 Secondary Protein Structure Prediction

In protein structure prediction, the elements of the sequence are the symbols representing one of the 20 amino acids. The 20 possible amino acids are akin to the vocabulary used in the text setting. Therefore, a one-hot encoding of the input is effective in these cases. Each position is associated with a class label corresponding to the secondary protein structure. This secondary structure can be either the alpha-helix, beta-sheet, or coil. Therefore, this problem can be reduced to token-level classification. A three-way softmax is used in the output layer. The work in [21] used a bidirectional recurrent neural network for prediction. This is because protein structure prediction is a problem that benefits from the context on both sides of a particular position. In general, the choice between using a uni-directional network and a bidirectional network is highly regulated by whether or not the prediction is causal to a historical segment or whether it depends on the context on both sides.

8.7.9 End-to-End Speech Recognition

In end-to-end speech recognition, one attempts to transcribe the raw audio files into character sequences while going through as few intermediate steps as possible. A small amount of preprocessing is still needed in order to make the data presentable as an input sequence. For example, the work in [164] presents the data as *spectrograms* derived from raw audio files using the *specgram* function of the *matplotlib* python toolkit. The width used was 254 Fourier windows with an overlap of 127 frames and 128 inputs per frame. The output is a character in the transcription sequence, which could include a character, a punctuation mark, a space, or even a null character. The label could be different depending on the application at hand. For example, the labels could be characters, phonemes, or musical notes. A bidirectional recurrent neural network is most appropriate to this setting, because the context on both sides of a character helps in improving accuracy.

One challenge associated with this type of setting is that we need the alignment between the frame representation of the audios and the transcription sequence. This type of alignment is not available a priori, and is in fact one of the outputs of the system. This leads to the problem of circular dependency between segmentation and recognition, which is also referred to as *Sayre's paradox*. This problem is solved with the use of *connectionist temporal classification*. In this approach, a dynamic programming algorithm [160] is combined with the (softmax) probabilistic outputs of the recurrent network in order to determine the alignment that maximizes the overall probability of generation. The reader is referred to [160, 164] for details.

8.7.10 Handwriting Recognition

A closely related application to speech recognition is that of handwriting recognition [161, 163]. In handwriting recognition, the input consists of a sequence of (x, y) coordinates, which represents the position of the tip of the pen at each time-stamp. The output corresponds

to a sequence of characters written by the pen. These coordinates are then used to extract further features such as a feature indicating whether the pen is touching the writing surface, the angles between nearby line segments, the velocity of the writing, and normalized values of the coordinates. The work in [161] extracts a total of 25 features. It is evident that multiple coordinates will create a character. However, it is hard to know exactly how many coordinates will create each character because it may vary significantly over the handwriting and style of different writers. Much like speech recognition, the issue of proper segmentation creates numerous challenges. This is the same Sayre's paradox that is encountered in speech recognition.

In unconstrained handwriting recognition, the handwriting contains a set of *strokes*, and by putting them together one can obtain characters. One possibility is to identify the strokes up front, and then use them to build characters. However, such an approach leads to inaccurate results, because the identification of stroke boundaries is an error-prone task. Since the errors tend to be additive over different phases, breaking up the task into separate stages is generally not a good idea. At a basic level, the task of handwriting recognition is no different from speech recognition. The only difference is in terms of the specific way in which the inputs and outputs are represented. As in the case of speech recognition, connectionist temporal classification is used in which a dynamic programming approach is combined with the softmax outputs of a recurrent neural network. Therefore, the alignment and the label-wise classification is performed simultaneously with dynamic programming in order to maximize the probability that a particular output sequence is generated for a particular input sequence. Readers are referred to [161, 163].

8.8 Summary

Recurrent neural networks are a class of neural networks that are used for sequence modeling. They can be expressed as time-layered networks in which the weights are shared between different layers. Recurrent neural networks can be hard to train, because they are prone to the vanishing and the exploding gradient problems. Some of these problems can be addressed with the use of enhanced training methods as discussed in Chapter 4. However, there are other ways of training more robust recurrent networks. A particular example that has found favor is the use of long short-term memory network. This network uses a gentler update process of the hidden states in order to avoid the vanishing and exploding gradient problems. Recurrent neural networks and their variants have found use in many applications such as image captioning, token-level classification, sentence classification, sentiment analysis, speech recognition, machine translation, and computational biology.

8.9 Bibliographic Notes and Software Resources

One of the earliest forms of the recurrent network was the Elman network [113]. This network was a precursor to modern recurrent networks. Werbos proposed the original version of backpropagation through time [545]. Another early algorithm for backpropagation in recurrent neural networks is provided in [389]. The vast majority of work on recurrent networks has been on symbolic data, although there is also some work on real-valued time series [83, 103, 583]. The regularization of recurrent neural networks is discussed in [577].

The effect of the spectral radius of the hidden-hidden matrix on the vanishing/exploding gradient problem is discussed in [231]. A detailed discussion of the exploding gradient

problem and other problems associated with recurrent neural networks may be found in [381, 382]. Recurrent neural networks (and their advanced variations) began to become more attractive after about 2010, when hardware advancements, increased data, and algorithmic tweaks made these methodologies far more attractive. The vanishing and exploding gradient problems in different types of deep networks, including recurrent networks, are discussed in [147, 215, 381]. The gradient clipping rule was discussed by Mikolov in his Ph.D. thesis [336]. The initialization of recurrent networks containing ReLUs is discussed in [277].

Early variants of the recurrent neural network included the echo-state network [230], which is also referred to as the *liquid-state machine* [315]. This paradigm is also referred to as *reservoir computing*. An overview of echo-state networks in the context of reservoir computing principles is provided in [310]. Teacher forcing methods are discussed in [107]. It has been shown in [15] that layer normalization provides better performance than batch normalization. Some related normalizations [303] can also be used for online learning [303].

The LSTM was first proposed in [214], and its use for language modeling is discussed in [492]. The challenges associated with training recurrent neural networks are discussed in [215, 381, 382]. Some of these problems can be addressed [338] by imposing constraints on the hidden-to-hidden matrix. Several variations of recurrent neural networks and LSTMs for language modeling are discussed in [72, 74, 158, 159, 325, 340]. Bidirectional recurrent neural networks are proposed in [454]. The particular discussion of LSTMs in this chapter is based on [158], and an alternative gated recurrent unit (GRU) is presented in [72, 74]. A guide to understanding recurrent neural networks is available in [243]. Further discussions on the sequence-centric and natural language applications of recurrent neural networks are available in [150, 307]. LSTM networks are also used for sequence labeling [157]. The use of a combination of convolutional neural networks and recurrent neural networks for image captioning is discussed in [236, 529]. Sequence-to-sequence learning methods for machine translation are discussed in [72, 241, 496, 556]. Bidirectional recurrent networks and LSTMs for protein structure prediction, handwriting recognition, translation, and speech recognition are discussed in [21, 161, 162, 164, 392, 493]. In recent years, neural networks have also been used in temporal collaborative filtering [481, 554, 584]. A generative model for dialogues with recurrent networks is discussed in [460, 461]. The use of recurrent neural networks for action recognition is discussed in [523]. Large-scaled pre-trained language models were proposed in ELMo [386], which can be used for a variety of downstream tasks.

Software Resources

Recurrent neural networks are supported by numerous software frameworks like *Caffe* [596], *Torch* [597], *Theano* [598], and *TensorFlow* [599]. The character-level RNN of this chapter [243, 640] is available at [604]. The *TensorFlow* implementation of ELMo is available at [678].

8.10 Exercises

1. Download the character-level RNN in [604], and train it on the "*tiny Shakespeare*" data set available at the same location. Create outputs of the language model after training for (i) 5 epochs, (ii) 50 epochs, and (iii) 500 epochs. What significant differences do you see between the three outputs?

2. Consider an echo-state network in which the hidden states are partitioned into K groups with p/K units each. The hidden states of a particular group are only allowed

to have connections within their own group in the next time-stamp. Discuss how this approach is related to an ensemble method in which K independent echo-state networks are constructed and the predictions of the K networks are averaged.

3. Show how you can modify the feed-forward architecture discussed in section 3.5 of Chapter 3 in order to create a recurrent neural network that can handle temporal recommender systems. Implement this adaptation and compare its performance to the feed-forward architecture on the Netflix prize data set.

4. Consider a recurrent network with two hidden states in each time layer. Every entry of the 2×2 matrix W_{hh} of transformations between hidden states is 3.5. The sigmoid activation is used between hidden states of different temporal layers. Would the network be more prone to the vanishing or to the exploding gradient problem?

5. Suppose that you have a large database of biological strings containing sequences of nucleobases drawn from $\{A, C, T, G\}$. Some of these strings contain unusual mutations representing changes in the nucleobases. Propose an unsupervised method (i.e., neural architecture) using RNNs in order to detect these mutations.

6. How would your architecture for Exercise 5 change if you were given a training database in which the mutation positions were tagged (but untagged test examples)?

7. Recommend possible methods for pre-training the input and output layers in the machine translation approach with sequence-to-sequence learning.

8. Consider a social network with a large volume of messages sent between sender-receiver pairs, and we are interested only in the messages containing an identifying keyword, referred to as a *hashtag*. Create a real-time model using an RNN, which has the capability to recommend hashtags of interest to each user, together with potential followers of that user who might be interested in messages related to that hashtag. Assume that you have enough computational resources to incrementally train an RNN.

9. If the training data set is re-scaled by a particular factor, do the learned weights of either batch normalization or layer normalization change? What would be your answer if only a small subset of points in the training data set are re-scaled? Would the learned weights in either normalization method be affected if the data set is re-centered?

10. Consider a setting in which you have a multilingual database of translated sentence-pairs. Although you have sufficient representation of each language, some *pairs* might not be well represented. Show how you can use this training data to (i) create the same universal code for a particular sentence across all languages, and (ii) have the ability to translate even between pairs of languages not well represented in the database.

11. Discuss how the neural architecture that performs language modeling (Figure 8.4) can be directly trained for machine translation by using appropriate training data.

12. Challenging: A recurrent neural network seems to ne inherently designed for 1-dimensional sequences. Can you think of a way of extending the recurrent neural network to 2-dimensional sequences? For example, a black-and-white image can be considered a two-dimensional sequence of pixel intensities. Propose the recurrence equation for the forward pass.

Chapter 9

Convolutional Neural Networks

"The soul never thinks without a picture."–Aristotle

9.1 Introduction

Convolutional neural networks are designed to work with grid-structured inputs, which have strong spatial dependencies in local regions of the grid. The most obvious example of grid-structured data is a 2-dimensional image. This type of data also exhibits spatial dependencies, because adjacent spatial locations in an image often have similar color values of the individual pixels. An additional dimension captures the different colors, which creates a 3-dimensional input *volume*. Therefore, the features in a convolutional neural network have dependencies among one another based on spatial distances. Other forms of sequential data like text, time-series, and sequences can also be considered special cases of grid-structured data with various types of relationships among adjacent items. Therefore, convolutional networks have also been occasionally applied to these types of data. One advantage of image data is that the effects of specific inputs on the feature representations can often be described in an intuitive way. Therefore, this chapter will primarily work with the image data setting. A brief discussion of other data domains will also be provided.

An important defining characteristic of convolutional neural networks is an operation, which is referred to as *convolution*. A convolution operation is a dot-product operation between a grid-structured set of weights and similar grid-structured inputs drawn from different spatial localities in the input volume. This type of operation is useful for data with a high level of spatial or other locality, such as image data. Therefore, convolutional neural networks are defined as networks that use the convolutional operation in at least one layer, although most convolutional neural networks use this operation in multiple layers.

9.1.1 Historical Perspective and Biological Inspiration

Convolutional neural networks were one of the first success stories of deep learning, well before recent advancements in training techniques led to improved performance in other types

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_9

of architectures. In fact, the eye-catching successes of some convolutional neural network architectures in image-classification contests after 2011 led to broader attention to the field of deep learning. Long-standing benchmarks like *ImageNet* [605] with a top-5 classification error-rate of more than 25% were brought down to less than 4% in the years between 2011 and 2015. Convolutional neural networks are well suited to the process of hierarchical feature engineering with depth; this is reflected in the fact that the deepest neural networks in all domains are drawn from the field of convolutional networks. Furthermore, these networks also represent excellent examples of how biologically inspired neural networks can sometimes provide ground-breaking results. The best convolutional neural networks today reach or exceed human-level performance, a feat considered impossible by most experts in computer vision only a couple of decades back.

The early motivation for convolutional neural networks was derived from experiments by Hubel and Wiesel on a cat's visual cortex [223]. The visual cortex has small regions of cells that are sensitive to specific regions in the visual field. In other words, if specific areas of the visual field are excited, then those cells in the visual cortex will be activated as well. Furthermore, the excited cells also depend on the shape and orientation of the objects in the visual field. For example, vertical edges cause some neuronal cells to be excited, whereas horizontal edges cause other neuronal cells to be excited. The cells are connected using a layered architecture, and this discovery led to the conjecture that mammals use these different layers to construct portions of images at different levels of abstraction. From a machine learning point of view, this principle is similar to that of hierarchical feature extraction. As we will see later, convolutional neural networks achieve something similar by encoding primitive shapes in earlier layers, and more complex shapes in later layers.

Based on these biological inspirations, the earliest neural model was the *neocognitron* [132]. Based on this architecture, one of the first convolutional architectures, referred to as *LeNet-5* [285], was developed. This network was used by banks to identify hand-written numbers on checks. Since then, the convolutional neural network has not evolved much; the main difference is in terms of using more layers and stable activation functions like the ReLU. Furthermore, numerous training tricks and powerful hardware options are available to achieve better success in training deep networks with larger data sets.

A factor that has played an important role in increasing the prominence of convolutional neural networks has been the annual *ImageNet* competition [606] (also referred to as *"ImageNet Large Scale Visual Recognition Challenge [ILSVRC]"*). The ILSVRC competition uses the *ImageNet* data set [605], which is discussed in section 1.7.2 of Chapter 1. Convolutional neural networks have been consistent winners of this contest since 2012, when *AlexNet* [263] was proposed. In fact, the dominance of convolutional neural networks for image classification is so well recognized today that almost all entries in recent editions of this contest have been convolutional neural networks. In spite of the fact that the vast majority of eye-catching performance gains have occurred from 2012 to 2015, the architectural differences between recent winners and some of the earliest convolutional neural networks are rather small at least at a conceptual level. Nevertheless, small details seem to matter a lot when working with almost all types of neural networks.

9.1.2 Broader Observations about Convolutional Neural Networks

The secret to the success of any neural architecture lies in designing the structure of the network with a semantic understanding of the domain at hand. Convolutional neural networks are heavily based on this principle, because they use sparse connections with a high-level of parameter-sharing in a domain-sensitive way. In other words, not all states in a particular

layer are connected to those in the previous layer in an indiscriminate way. Rather, the value of a feature in a particular layer is connected only to a local spatial region in the previous layer with a consistent set of shared parameters across the full spatial footprint of the image. This type of architecture can be viewed as a domain-aware regularization, which was derived from the biological insights in Hubel and Wiesel's early work. In general, the success of the convolutional neural network has important lessons for other data domains. A carefully designed architecture, in which the relationships and dependencies among the data items are used in order to reduce the parameter footprint, provides the key to high accuracy.

A significant level of domain-aware regularization is also available in recurrent neural networks, which share the parameters from different temporal periods. This sharing is based on the assumption that temporal dependencies remain invariant with time. Recurrent neural networks are based on intuitive understanding of temporal relationships, whereas convolutional neural networks are based on an intuitive understanding of spatial relationships. The latter intuition was directly extracted from the organization of biological neurons in a cat's visual cortex. This outstanding success provides a motivation to explore how neuroscience may be leveraged to design neural networks in clever ways. Even though artificial neural networks are only caricatures of the true complexity of the biological brain, one should not underestimate the intuition that one can obtain by studying the basic principles of neuroscience [186].

Chapter Organization

This chapter is organized as follows. The next section will introduce the basics of a convolutional neural network, the various operations, and the way in which they are organized. The training process for convolutional networks is discussed in section 9.3. Case studies with some typical convolutional neural networks that have won recent competitions are discussed in section 9.4. The convolutional autoencoder is discussed in section 9.5. A variety of applications of convolutional networks are discussed in section 9.6. A summary is given in section 9.7.

9.2 The Basic Structure of a Convolutional Network

In convolutional neural networks, the states in each layer are arranged according to a spatial grid structure. These spatial relationships are inherited from one layer to the next because each feature value is based on a small local spatial region in the previous layer. It is important to maintain these spatial relationships among the grid cells, because the convolution operation and the transformation to the next layer is critically dependent on these relationships. Each layer in the convolutional network is a 3-dimensional grid structure, which has a *height*, *width*, and *depth*. The depth of a layer in a convolutional neural network should not be confused with the depth of the network itself. The word "depth" (when used in the context of a single layer) refers to the number of *channels* in each layer, such as the number of primary color channels (e.g., blue, green, and red) in the input image or the number of feature maps in the hidden layers. The use of the word "depth" to refer to both the number of feature maps in each layer as well as the number of layers is an unfortunate overloading of terminology used in convolutional networks, but we will be careful while using this term, so that it is clear from its context.

The convolutional neural network functions much like a traditional feed-forward neural network, except that the operations in its layers are spatially organized with sparse (and

carefully designed) connections between layers. The three types of layers that are commonly present in a convolutional neural network are *convolution*, *pooling*, and *ReLU*. The ReLU activation is no different from a traditional neural network. In addition, a final set of layers is often fully connected and maps in an application-specific way to a set of output nodes. In the following, we will describe each of the different types of operations and layers, and the typical way in which these layers are interleaved in a convolutional neural network.

Why do we need depth in each layer of a convolutional neural network? To understand this point, let us examine how the input to the convolutional neural network is organized. The input data to the convolutional neural network is organized into a 2-dimensional grid structure, and the values of the individual grid points are referred to as *pixels*. Each pixel, therefore, corresponds to a spatial location within the image. However, in order to encode the precise color of the pixel, we need a multidimensional array of values at each grid location. In the RGB color scheme, we have an intensity of the three primary colors, corresponding to red, green, and blue, respectively. Therefore, if the spatial dimensions of an image are 32×32 pixels and the depth is 3 (corresponding to the RGB color channels), then the overall number of pixels in the image is $32 \times 32 \times 3$. This particular image size is quite common, and also occurs in a popularly used data set for benchmarking, known as CIFAR-10 [607]. An example of this organization is shown in Figure 9.1(a). It is natural to represent the input layer in this 3-dimensional structure because two dimensions are devoted to spatial relationships and a third dimension is devoted to the independent properties along these channels. For example, the intensities of the primary colors are the independent properties in the first layer. In the hidden layers, these independent properties correspond to various types of shapes extracted from local regions of the image. For the purpose of discussion, assume that the input in the qth layer is of size $L_q \times B_q \times d_q$. Here, L_q refers to the *height* (or length), B_q refers to the width (or breadth), and d_q is the depth. In almost all image-centric applications, the values of L_q and B_q are the same. However, we will work with separate notations for height and width in order to retain generality in presentation.

For the first (input) layer, these values are decided by the nature of the input data and its preprocessing. In the above example, the values are $L_1 = 32$, $B_1 = 32$, and $d_1 = 3$. Later layers have exactly the same 3-dimensional organization, except that each of the d_q 2-dimensional grid of values for a particular input can no longer be considered a grid of raw pixels. Furthermore, the value of d_q is much larger than three for the hidden layers because the number of independent properties of a given local region that are relevant to classification can be quite significant. For $q > 1$, these grids of values are referred to as *feature maps* or *activation maps*. These values are analogous to the values in the hidden layers in a feed-forward network.

In the convolutional neural network, the parameters are organized into sets of 3-dimensional structural units, known as *filters* or *kernels*. The filter is usually square in terms of its spatial dimensions, which are typically much smaller than those of the layer the filter is applied to. On the other hand, *the depth of a filter is always same is the same as that of the layer to which it is applied*. Assume that the dimensions of the filter in the qth layer are $F_q \times F_q \times d_q$. An example of a filter with $F_1 = 5$ and $d_1 = 3$ is shown in Figure 9.1(a). It is common for the value of F_q to be small and odd. Examples of commonly used values of F_q are 3 and 5, although there are some interesting cases in which it is possible to use $F_q = 1$.

The *convolution operation* places the filter at each possible position in the image (or hidden layer) so that the filter fully overlaps with the image, and performs a dot product between the $F_q \times F_q \times d_q$ parameters in the filter and the matching grid in the input volume (with same size $F_q \times F_q \times d_q$). The dot product is performed by treating the entries

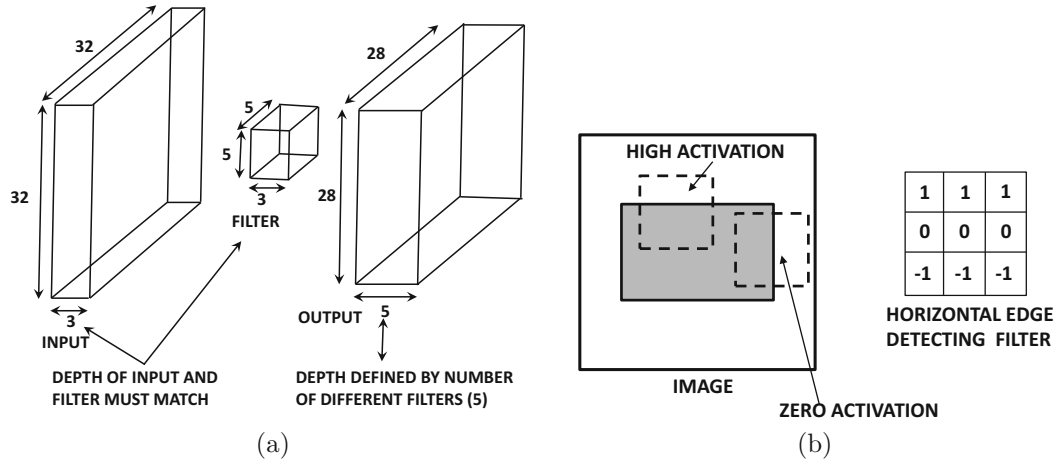


Figure 9.1: (a) The convolution between an input layer of size $32 \times 32 \times 3$ and a filter of size $5 \times 5 \times 3$ produces an output layer with spatial dimensions 28×28. The depth of the resulting output depends on the number of distinct filters and not on the dimensions of the input layer or filter. (b) Sliding a filter around the image tries to look for a particular feature in various windows of the image.

in the relevant 3-dimensional region of the input volume and the filter as vectors of size $F_q \times F_q \times d_q$, so that the elements in both vectors are ordered based on their corresponding positions in the grid-structured volume. How many possible positions are there for placing the filter? This question is important, because each such position therefore defines a spatial "pixel" (or, more accurately, a *feature*) in the next layer. In other words, the number of alignments between the filter and image defines the spatial height and width of the next hidden layer. The relative spatial positions of the features in the next layer are defined based on the relative positions of the upper left corners of the corresponding spatial grids in the previous layer. When performing convolutions in the qth layer, one can align the filter at $L_{q+1} = (L_q - F_q + 1)$ positions along the height and $B_{q+1} = (B_q - F_q + 1)$ along the width of the image (without having a portion of the filter "sticking out" from the borders of the image). This results in a total of $L_{q+1} \times B_{q+1}$ possible dot products, which defines the size of the next hidden layer. In the previous example, the values of L_2 and B_2 are therefore defined as follows:

$$L_2 = 32 - 5 + 1 = 28$$
$$B_2 = 32 - 5 + 1 = 28$$

The next hidden layer of size 28×28 is shown in Figure 9.1(a). However, this hidden layer also has a depth of size $d_2 = 5$. Where does this depth come from? This is achieved by using 5 different filters with their own independent sets of parameters. Each of these 5 sets of spatially arranged features obtained from the output of a single filter is referred to as a *feature map*. Clearly, an increased number of feature maps is a result of a larger number of filters (i.e., parameter footprint), which is $F_q^2 \cdot d_q \cdot d_{q+1}$ for the qth layer. *The number of filters used in each layer controls the capacity of the model because it directly controls the number of parameters.* Furthermore, increasing the number of filters in a particular layer increases the number of feature maps (i.e., depth) of the next layer. It is possible for different layers to have very different numbers of feature maps, depending on the number of filters we use

for the convolution operation in the previous layer. For example, the input layer typically only has three color channels, but it is possible for the each of the later hidden layers to have depths (i.e., number of feature maps) of more than 500. The idea here is that each filter tries to identify a particular type of spatial pattern in a small rectangular region of the image, and therefore a large number of filters is required to capture a broad variety of the possible shapes that are combined to create the final image (unlike the case of the input layer, in which three RGB channels are sufficient). Typically, the later layers tend to have a smaller spatial footprint, but greater depth in terms of the number of feature maps. For example, the filter shown in Figure 9.1(b) represents a horizontal edge detector on a grayscale image with one channel. As shown in Figure 9.1(b), the resulting feature will have high activation at each position where a horizontal edge is seen. A perfectly vertical edge will give zero activation, whereas a slanted edge might give intermediate activation. Therefore, sliding the filter everywhere in the image will already detect several key outlines of the image in a single feature map of the output volume. Multiple filters are used to create an output volume with more than one feature map. For example, a different filter might create a spatial feature map of vertical edge activations.

We are now ready to formally define the convolution operation. The pth filter in the qth layer has parameters denoted by the 3-dimensional tensor $W^{(p,q)} = [w_{ijk}^{(p,q)}]$. The indices i, j, k indicate the positions along the height, width, and depth of the filter. The feature maps in the qth layer are represented by the 3-dimensional tensor $H^{(q)} = [h_{ijk}^{(q)}]$. When the value of q is 1, the special case corresponding to the notation $H^{(1)}$ simply represents the input layer (which is not hidden). Then, the convolutional operations from the qth layer to the $(q+1)$th layer are defined as follows:

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \qquad \forall i \in \{1 \ldots, L_q - F_q + 1\}$$
$$\forall j \in \{1 \ldots B_q - F_q + 1\}$$
$$\forall p \in \{1 \ldots d_{q+1}\}$$

The expression above seems notationally complex, although the underlying convolutional operation is really a simple dot product over the entire volume of the filter, which is repeated over all valid spatial positions (i, j) and filters (indexed by p). It is intuitively helpful to understand a convolution operation by placing the filter at each of the 28×28 possible spatial positions in the first layer of Figure 9.1(a) and performing a dot product between the vector of $5 \times 5 \times 3 = 75$ values in the filter and the corresponding 75 values in $H^{(1)}$. Even though the size of the input layer in Figure 9.1(a) is 32×32, there are only $(32-5+1) \times (32-5+1)$ possible spatial alignments between an input volume of size 32×32 and a filter of size 5×5.

The convolution operation brings to mind Hubel and Wiesel's experiments that use the activations in small regions of the visual field to activate particular neurons. In the case of convolutional neural networks, this visual field is defined by the filter, which is applied to all locations of the image in order to detect the presence of a shape at each spatial location. Furthermore, the filters in earlier layers tend to detect more primitive shapes, whereas the filters in later layers create more complex compositions of these primitive shapes. This is not particularly surprising because most deep neural networks are good at hierarchical feature engineering.

One property of convolution is that it shows *equivariance to translation*. In other words, if we shifted the pixel values in the input in any direction by one unit and then applied convolution, the corresponding feature values will shift with the input values. This is because

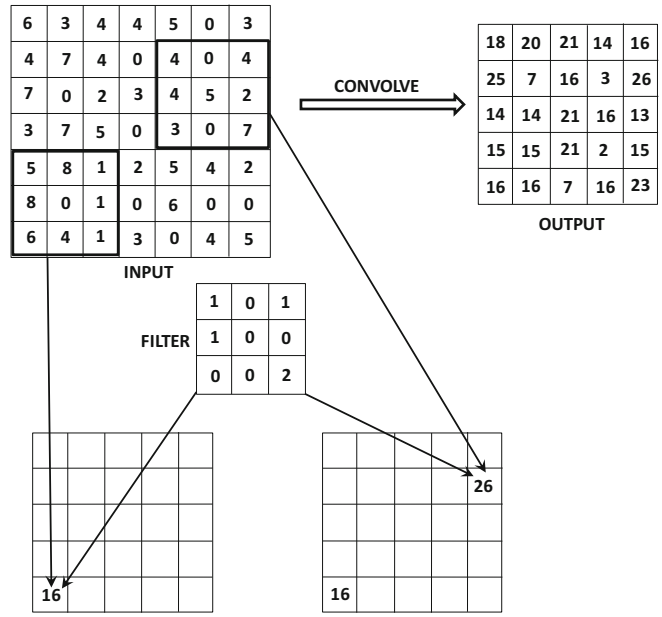


Figure 9.2: An example of a convolution between a $7 \times 7 \times 1$ input and a $3 \times 3 \times 1$ filter with stride of 1. A depth of 1 has been chosen for the filter/input for simplicity. For depths larger than 1, the contributions of each input feature map will be added to create a single value in the feature map. A single filter will always create a single feature map irrespective of its depth.

of the shared parameters of the filter across the entire convolution. The reason for sharing parameters across the entire convolution is that the presence of a particular shape in any part of the image should be processed in the same way irrespective of its specific spatial location.

In the following, we provide an example of the convolution operation. In Figure 9.2, we have shown an example of an input layer and a filter with depth 1 for simplicity (which does occur in the case of grayscale images with a single color channel). Note that the depth of a layer must exactly match that of its filter/kernel, and the contributions of the dot products over all the feature maps in the corresponding grid region of a particular layer will need to be added (in the general case) to create a single output feature value in the next layer. Figure 9.2 depicts two specific examples of the convolution operations with a layer of size $7 \times 7 \times 1$ and a $3 \times 3 \times 1$ filter in the bottom row. Furthermore, the entire feature map of the next layer is shown on the upper right-hand side of Figure 9.2. Examples of two convolution operations are shown in which the outputs are 16 and 26, respectively. These values are arrived at by using the following multiplication and aggregation operations:

$$5 \times 1 + 8 \times 1 + 1 \times 1 + 1 \times 2 = 16$$
$$4 \times 1 + 4 \times 1 + 4 \times 1 + 7 \times 2 = 26$$

The multiplications with zeros have been omitted in the above aggregation. In the event that the depths of the layer and its corresponding filter are greater than 1, the above operations are performed for each spatial map and then aggregated across the entire depth of the filter.

A convolution in the qth layer increases the *receptive field* of a feature from the qth layer to the $(q+1)$th layer. In other words, each feature in the next layer captures a larger spatial region in the input layer. For example, when using a 3×3 filter convolution successively in three layers, the activations in the first, second, and third hidden layers capture pixel regions of size 3×3, 5×5, and 7×7, respectively, in the *original input image*. As we will see later, other types of operations increase the receptive fields further, as they reduce the size of the spatial footprint of the layers. This is a natural consequence of the fact that features in later layers capture complex characteristics of the image over larger spatial regions, and then combine the simpler features in earlier layers.

When performing the operations from the qth layer to the $(q+1)$th layer, the depth d_{q+1} of the computed layer depends on the *number* of filters in the qth layer, and it is independent of the *depth* of the qth layer or any of its other dimensions. In other words, the depth d_{q+1} in the $(q+1)$th layer is always equal to the number of filters in the qth layer. For example, the depth of the second layer in Figure 9.1(a) is 5, because a total of five filters are used in the first layer for the transformation. However, in order to perform the convolutions in the second layer (to create the third layer), one must now use filters of depth 5 in order to match the new depth of this layer, even though filters of depth 3 were used in the convolutions of the first layer (to create the second layer).

9.2.1 Padding

One observation is that the convolution operation reduces the size of the $(q+1)$th layer in comparison with the size of the qth layer. This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image (or of the feature map, in the case of hidden layers). This problem can be resolved by using *padding*. In padding, one adds $(F_q - 1)/2$ "pixels" all around the borders of the feature map in order to maintain the spatial footprint. Note that these pixels are really feature values in the case of padding hidden layers. The value of each of these padded feature values is set to 0, irrespective of whether the input or the hidden layers are being padded. As a result, the spatial height and width of the input volume will both increase by $(F_q - 1)$, which is exactly what they reduce by (in the output volume) after the convolution is performed. The padded portions do not contribute to the final dot product because their values are set to 0. In a sense, what padding does is to allow the convolution operation with a portion of the filter "sticking out" from the borders of the layer and then performing the dot product only over the portion of the layer where the values are defined. This type of padding is referred to as *half-padding* because (almost) half the filter is sticking out from all sides of the spatial input in the case where the filter is placed in its extreme spatial position along the edges. Half-padding is designed to maintain the spatial footprint exactly.

When padding is not used, the resulting "padding" is also referred to as a *valid padding*. Valid padding generally does not work well from an experimental point of view. Using half-padding ensures that some of the critical information at the borders of the layer is represented in a standalone way. In the case of valid padding, the contributions of the pixels on the borders of the layer will be under-represented compared to the central pixels in the next hidden layer, which is undesirable. Furthermore, this under-representation will be compounded over multiple layers. Therefore, padding is typically performed in all layers, and not just in the first layer where the spatial locations correspond to input values. Consider a situation in which the layer has size $32 \times 32 \times 3$ and the filter is of size $5 \times 5 \times 3$. Therefore, $(5-1)/2 = 2$ zeros are padded on all sides of the image. As a result, the 32×32 spatial footprint first increases to 36×36 because of padding, and then it reduces back to 32×32

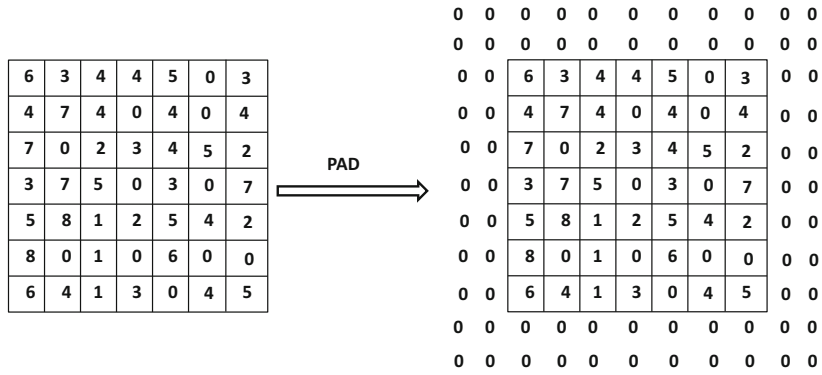


Figure 9.3: An example of padding. Each of the d_q activation maps in the entire depth of the qth layer are padded in this way.

after performing the convolution. An example of the padding of a single feature map is shown in Figure 9.3, where two zeros are padded on all sides of the image (or feature map). This is a similar situation as discussed above (in terms of addition of two zeros), except that the spatial dimensions of the image are much smaller than 32×32 in order to enable illustration in a reasonable amount of space.

Another useful form of padding is *full-padding*. In full-padding, we allow (almost) the *full* filter to stick out from various sides of the input. In other words, a portion of the filter of size $F_q - 1$ is allowed to stick out from any side of the input with an overlap of only one spatial feature. For example, the kernel and the input image might overlap at a single pixel at an extreme corner. Therefore, the input is padded with $(F_q - 1)$ zeros on each side. In other words, each spatial dimension of the input increases by $2(F_q - 1)$. Therefore, if the input dimensions in the original image are L_q and B_q, the padded spatial dimensions in the input volume become $L_q + 2(F_q - 1)$ and $B_q + 2(F_q - 1)$. After performing the convolution, the feature-map dimensions in layer $(q+1)$ become $L_q + F_q - 1$ and $B_q + F_q - 1$, respectively. While convolution normally reduces the spatial footprint, full padding *increases* the spatial footprint. Interestingly, full-padding increases each dimension of the spatial footprint by the same value $(F_q - 1)$ that no-padding decreases it. *This relationship is not a coincidence because a "reverse" convolution operation can be implemented by applying another convolution on the fully padded output (of the original convolution) with an appropriately defined kernel of the same size.* This type of "reverse" convolution occurs frequently in the back-propagation and autoencoder algorithms for convolutional neural networks. Fully padded inputs are useful because they increase the spatial footprint, which is required in several types of convolutional autoencoders.

9.2.2 Strides

There are other ways in which convolution can reduce the spatial footprint of the image (or hidden layer). The above approach performs the convolution at every position in the spatial location of the feature map. However, it is not necessary to perform the convolution at every spatial position in the layer. One can reduce the level of granularity of the convolution by using the notion of *strides*. The description above corresponds to the case when a stride of 1 is used. When a stride of S_q is used in the qth layer, the convolution is performed at the locations 1, $S_q + 1$, $2\,S_q + 1$, and so on along both spatial dimensions of the layer. The

spatial size of the output on performing this convolution[1] has height of $(L_q - F_q)/S_q + 1$ and a width of $(B_q - F_q)/S_q + 1$. As a result, the use of strides will result in a reduction of each spatial dimension of the layer by a factor of approximately S_q and the area by S_q^2, although the actual factor may vary because of edge effects. It is most common to use a stride of 1, although a stride of 2 is occasionally used as well. It is rare to use strides more than 2 in normal circumstances. Even though a stride of 4 was used in the input layer of the winning architecture [263] of the ILSVRC competition of 2012, the winning entry in the subsequent year reduced the stride to 2 [581] to improve accuracy. Larger strides can be helpful in memory-constrained settings or to reduce overfitting if the spatial resolution is unnecessarily high. Strides have the effect of rapidly increasing the receptive field of each feature in the hidden layer, while reducing the spatial footprint of the entire layer. An increased receptive field is useful in order to capture a complex feature in a larger spatial region of the image. As we will see later, the hierarchical feature engineering process of a convolutional neural network captures more complex shapes in later layers. Historically, the receptive fields have been increased with another operation, known as the *max-pooling* operation. In recent years, larger strides have been used in lieu [194, 482] of max-pooling operations, which will be discussed later.

Typical Settings

It is common to use stride sizes of 1, and occasionally stride sizes of 2. Furthermore, it is common to use square images with $L_q = B_q$. In cases where the input images are not square, preprocessing is used to enforce this property. For example, one can extract square patches of the image to create the training data. The number of filters in each layer is often a power of 2, because this often results in more efficient processing. Such an approach also leads to hidden layer depths that are powers of 2. Typical values of the spatial extent of the filter size (denoted by F_q) are 3 or 5. In general, small filter sizes often provide the best results, although some practical challenges exist in using filter sizes that are too small. Small filter sizes often tend to be more powerful, because they lead to deeper networks (for the same parameter footprint). In fact, one of the top entries in an ILSVRC contest, referred to as *VGG* [474], was the first to experiment with a filter dimension of only $F_q = 3$ for all layers, and the approach was found to work very well in comparison with larger filter sizes.

Use of Bias

As in all neural networks, it is also possible to add biases to the forward operations. Each unique filter in a layer is associated with its own bias. Therefore, the pth filter in the qth layer has bias $b^{(p,q)}$. When any convolution is performed with the pth filter in the qth layer, the value of $b^{(p,q)}$ is added to the dot product. The use of the bias simply increases the number of parameters in each filter by 1, and therefore it is not a significant overhead. Like all other parameters, the bias is learned during backpropagation. One can treat the bias as a weight of a connection whose input is always set to +1. This special input is used in all convolutions, irrespective of the spatial location of the convolution. Therefore, one can assume that a special pixel appears in the input whose value is always set to 1. Therefore, the number of input features in the qth layer is $1 + L_q \times B_q \times d_q$. This is a standard feature-engineering trick that is used for handling bias in all forms of machine learning.

[1] Here, it is assumed that $(L_q - F_q)$ is exactly divisible by S_q in order to obtain a clean fit of the convolution filter with the original image. Otherwise, some ad hoc modifications are needed to handle edge effects. In general, this is not a desirable solution.

9.2.3 The ReLU Layer

The convolution operation is interleaved with the pooling and ReLU operations. The ReLU activation is not very different from how it is applied in a traditional neural network. For each of the $L_q \times B_q \times d_q$ values in a layer, the ReLU activation function is applied to it to create $L_q \times B_q \times d_q$ thresholded values. These values are then passed on to the next layer. Therefore, applying the ReLU does not change the dimensions of a layer because it is a simple one-to-one mapping of activation values. In traditional neural networks, the activation function is combined with a linear transformation with a matrix of weights to create the next layer of activations. Similarly, a ReLU typically follows a convolution operation (which is the rough equivalent of the linear transformation in traditional neural networks), and the ReLU layer is often not explicitly shown in pictorial illustrations of most convolution neural network architectures.

It is noteworthy that the use of the ReLU activation function is a recent evolution, and ir replaced the use of saturating activation functions like sigmoid and tanh. It was shown in [263] that the use of the ReLU has tremendous advantages over these activation functions both in terms of speed and accuracy. Increased speed is also connected to accuracy because it allows one to use deeper models and train them for a longer time. In recent years, the use of the ReLU activation function has replaced the other activation functions in convolutional neural network design to an extent that this chapter will simply use the ReLU as the default activation function (unless otherwise mentioned).

9.2.4 Pooling

The pooling operation is, however, quite different. The pooling operation works on small grid regions of size $P_q \times P_q$ in each layer, and produces another layer *with the same depth* (unlike filters). For each square region of size $P_q \times P_q$ in each of the d_q activation maps, the *maximum* of these values is returned. This approach is referred to as *max-pooling*. If a stride of 1 is used, then this will produce a new layer of size $(L_q - P_q + 1) \times (B_q - P_q + 1) \times d_q$. However, it is more common to use a stride $S_q > 1$ in pooling. In such cases, the length of the new layer will be $(L_q - P_q)/S_q + 1$ and the breadth will be $(B_q - P_q)/S_q + 1$. Therefore, pooling drastically reduces the spatial dimensions of each activation map (thereby enabling efficient learning). The max-pooling layers are interleaved with the convolutional/ReLU layers, although max-pooling occurs after a few consecutive convolutional/ReLU layers in deep architectures. This is because pooling drastically reduces the spatial size of the feature map, and only a few pooling operations are required to reduce the spatial map to a small constant size.

Unlike with convolution operations, pooling is done at the level of *each* activation map. Whereas a convolution operation simultaneously uses all d_q feature maps in combination with a filter to produce a single feature value, pooling independently operates on each feature map to produce another feature map. Therefore, the operation of pooling does not change the number of feature maps. In other words, the depth of the layer created using pooling is the same as that of the layer on which the pooling operation was performed. Examples of pooling with strides of 1 and 2 are shown in Figure 9.4. Here, we use pooling over 3×3 regions. The typical size P_q of the region over which one performs pooling is 2×2. At a stride of 2, there would be no overlap among the different regions being pooled, and it is quite common to use this setting to reduce the spatial footprint of the activation maps.

Another type of pooling that is less commonly used is average pooling, which averages the pixels instead of using the maximum. In the earliest convolutional network, referred to

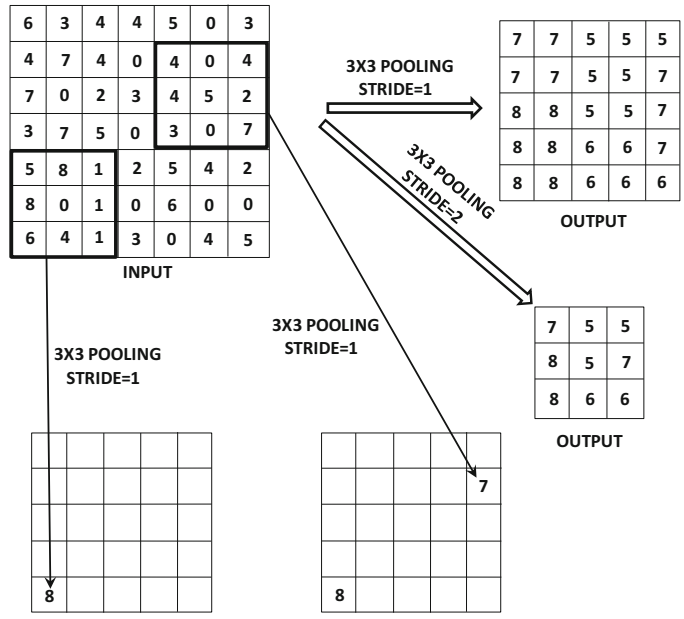


Figure 9.4: An example of a max-pooling of one activation map of size 7×7 with strides of 1 and 2. A stride of 1 creates a 5×5 activation map with heavily repeating elements because of maximization in overlapping regions. A stride of 2 creates a 3×3 activation map with less overlap. Unlike convolution, each activation map is independently processed and therefore the number of output activation maps is exactly equal to the number of input activation maps.

as *LeNet-5*, a variant of average pooling was used and was referred to as *subsampling*. Max-pooling picks out the dominant features from the image by picking the brightest "pixels," (e.g., sharp edges in dark backgrounds), whereas average pooling smooths out the features by averaging over a region and simply reduces resolution. In modern usage, subsampling refers to replacement of pixels in a grid by a single pixel (which also serves the purpose of resolution reduction).

Both pooling operators reduce spatial footprint and enable efficient training. However, max-pooling also serves an important purpose of preserving *translation invariance*. The idea is that a banana has the same interpretation, whether it is at the top or the bottom of an image. This property helps in being able to consistently classify images with different relative positions of objects in an accurate way. Max-pooling results in (some) invariance to translation because shifting the image reduces the shift in the activation map after pooling. However, average pooling does not address translation invariance, and can, in fact, be implemented as convolution operations with multiple filters of fixed weights (see Exercise 11). Therefore, the choice of pooling operator should also depend on the importance of relative positions of objects in the image.

Another important purpose of pooling is that it increases the size of the receptive field while reducing the spatial footprint of the layer because of the use of strides larger than 1. Increased sizes of receptive fields are needed to be able to capture larger regions of the image within a complex feature in later layers. Most of the rapid reductions in spatial footprints of the layers (and corresponding increases in receptive fields of the features) are caused

by the pooling operations. Convolutions increase the receptive field only gently unless the stride is larger than 1. In recent years, it has been suggested that pooling is not always necessary. One can design a network with only convolutional and ReLU operations, and obtain the expansion of the receptive field by using larger strides within the convolutional operations [194, 482]. Therefore, there is an emerging trend in recent years to get rid of the max-pooling layers altogether. However, this trend has not been fully validated as of the writing of this book. While it is possible to see why strided convolutions might replace average pooling, max-pooling serves purposes beyond spatial footprint reduction. Max-pooling introduces nonlinearity and a greater amount of translation invariance, as compared to strided convolutions. Although nonlinearity can be achieved with the ReLU activation function, the translation effects of max-pooling cannot be exactly replicated by strided convolutions. At the very least, the two operations are not fully interchangeable.

9.2.5 Fully Connected Layers

Each feature in the final spatial layer is connected to each hidden state in the first fully connected layer. This layer functions in exactly the same way as a traditional feed-forward network. The connections among these layers are exactly structured like a traditional feed-forward network, and usually have multiple layers. Since the fully connected layers are densely connected, the vast majority of parameters lie in the fully connected layers. For example, if each of two fully connected layers has 4096 hidden units, then the connections between them have more than 16 million weights. Similarly, the connections from the last spatial layer to the first fully connected layer will have a large number of parameters. Even though the convolutional layers have a larger number of *activations* (and a larger memory footprint), the fully connected layers often have a larger number of *connections* (and parameter footprint). The reason that activations contribute to the memory footprint more significantly is that the number of activations are multiplied by mini-batch size while tracking variables in the forward and backward passes of backpropagation. These trade-offs are useful to keep in mind while choosing neural-network design based on specific types of resource constraints (e.g., data versus memory availability). The output layer of a convolutional neural network is designed in an application-specific way. In the particular cases of classification and regression, the final fully connected layer might use the logistic, softmax, or linear activation.

One alternative to using fully connected layers is to use average pooling across the whole spatial area of the final set of activation maps to create a single value. Therefore, the number of features created in the final spatial layer will be exactly equal to the number of filters. In this scenario, if the final activation maps are of size $7 \times 7 \times 256$, then 256 features will be created. Each feature will be the result of aggregating 49 values. This type of approach greatly reduces the parameter footprint of the fully connected layers, and it has some advantages in terms of generalizability. This approach was used in *GoogLeNet* [501]. In some applications like image segmentation, each pixel is associated with a class label, and one does not use fully connected layers. Fully convolutional networks with 1×1 convolutions are used in order to create an output spatial map.

9.2.6 The Interleaving between Layers

The convolution, pooling, and ReLU layers are typically interleaved in a neural network in order to increase the expressive power of the network. The ReLU layers often follow the convolutional layers, just as a nonlinear activation function typically follows the linear dot

product in traditional neural networks. Therefore, the convolutional and ReLU layers are typically stuck together one after the other. Some pictorial illustrations of neural architectures like *AlexNet* [263] do not explicitly show the ReLU layers because they are assumed to be always stuck to the end of the linear convolutional layers. After two or three sets of convolutional-ReLU combinations, one might have a max-pooling layer. Examples of this basic pattern are as follows:

CRCRP

CRCRCRP

Here, the convolutional layer is denoted by C, the ReLU layer is denoted by R, and the max-pooling layer is denoted by P. This entire pattern (including the max-pooling layer) might be repeated a few times in order to create a deep neural network. For example, if the first pattern above is repeated three times and followed by a fully connected layer (denoted by F), then we have the following neural network:

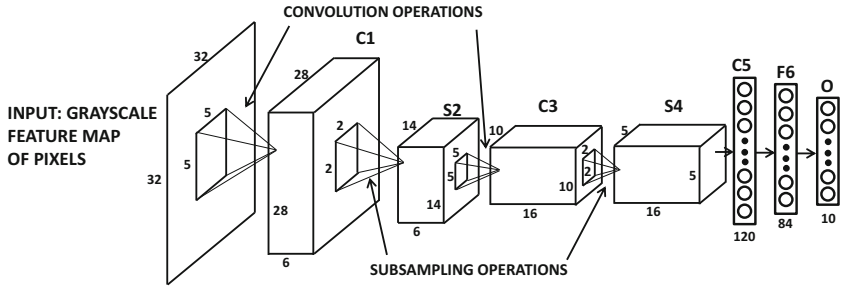
CRCRPCRCRPCRCRPF

The description above is not complete because one needs to specify the number/size/padding of filters/pooling layers. The pooling layer is the key step that tends to reduce the spatial footprint of the activation maps because it uses strides that are larger than 1. It is also possible to reduce the spatial footprints with strided convolutions instead of max-pooling. These networks are often quite deep, and it is not uncommon to have convolutional networks with more than 15 layers. Recent architectures also use *skip connections* between layers, which become increasingly important as the depth of the network increases (cf. section 9.4.5).

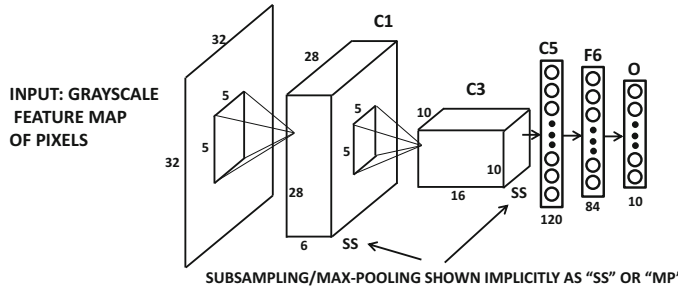
LeNet-5

Early networks were quite shallow. An example of one of the earliest neural networks is *LeNet-5* [285]. The input data is in grayscale, and there is only one color channel. The input is assumed to be the ASCII representation of a character. For the purpose of discussion, we will assume that there are ten types of characters (and therefore 10 outputs), although the approach can be used for any number of classes.

The network contained two convolution layers, two pooling layers, and three fully connected layers at the end. However, later layers contain multiple feature maps because of the use of multiple filters in each layer. The architecture of this network is shown in Figure 9.5. The first fully connected layer was also referred to as a convolution layer (labeled as $C5$) in the original work because the ability existed to generalize it to spatial features for larger input maps. However, the specific implementation of *LeNet-5* really used $C5$ as a fully connected layer, because the filter spatial size was the same as the input spatial size. This is why we are counting $C5$ as a fully connected layer in this exposition. It is noteworthy that two versions of *LeNet-5* are shown in Figures 9.5(a) and (b). The upper diagram of Figure 9.5(a) explicitly shows the subsampling layers, which is a variant of average pooling after multiplying with a trainable coefficient and adding a trainable bias. Modern architectural diagrams like *AlexNet* [263] often do not show the subsampling or max-pooling layers explicitly in order to accommodate the large number of layers. Such a concise architecture for *LeNet-5* is illustrated in Figure 9.5(b). The activation function layers are also not explicitly shown in either figure. In the original work in *LeNet-5*, the sigmoid activation function



(a) Detailed architectural representation



(a) Concise architectural representation

Figure 9.5: LeNet-5: One of the earliest convolutional neural networks.

occurs immediately after the subsampling operations, although this ordering is relatively unusual in recent architectures. In most modern architectures, subsampling is replaced by max-pooling, and the max-pooling layers occur less frequently than the convolution layers. Furthermore, the activations are typically performed immediately after each convolution (rather than after each max-pooling).

The number of layers in the architecture is often counted in terms of the number of layers with weighted spatial filters and the number of fully connected layers. In other words, subsampling/max-pooling and activation function layers are often not counted separately. The subsampling in *LeNet-5* used 2×2 spatial regions with stride 2. Furthermore, unlike max-pooling, the values were averaged, scaled with a trainable weight and then a bias was added. In modern architectures, the linear scaling and bias addition operations have been dispensed with. The concise architectural representation of Figure 9.5(b) is sometimes confusing to beginners because it is missing details such as the size of the max-pooling/subsampling filters. In fact, there is no unique way of representing these architectural details, and many variations are used by different authors. This chapter will show several such examples in the case studies.

This network is extremely shallow by modern standards; yet the basic principles have not changed since then. The main difference is that the ReLU activation had not appeared at that point, and sigmoid activation was often used in the earlier architectures. Furthermore, the use of average pooling is extremely uncommon today compared to max-pooling. Recent years have seen a move away from both max-pooling and subsampling, with strided convolutions as the preferred choice. *LeNet-5* also used ten radial basis function (RBF) units in the final layer (cf. Chapter 6), in which the prototype of each unit was compared to its input vector and the squared Euclidean distance between them was output. This is the

same as using the negative log-likelihood of the Gaussian distribution represented by that RBF unit. The parameter vectors of the RBF units were chosen by hand, and correspond to a stylized 7×12 bitmap image of the corresponding character class, which were flattened into a $7 \times 12 = 84$-dimensional representation. Note that the size of the penultimate layer is exactly 84 in order to enable the computation of the Euclidean distance between the vector corresponding to that layer and the parameter vector of the RBF unit. The ten outputs in the final layer provide the scores of the classes, and the smallest score among the ten units provides the prediction. This type of use of RBF units is now anachronistic in modern convolutional network design, and one generally tends to work with softmax units with log-likelihood loss on multinomial label outputs. *LeNet-5* was used extensively for character recognition, and was used by many banks to read checks.

9.2.7 Hierarchical Feature Engineering

It is instructive to examine the activations of the filters created by real-world images in different layers. In section 9.5, we will discuss a concrete way in which the features extracted in various layers can be visualized. For now, we provide a subjective interpretation. The activations of the filters in the early layers are low-level features like edges, whereas those in later layers put together these low-level features. For example, a mid-level feature might put together edges to create a hexagon, whereas a higher-level feature might put together the mid-level hexagons to create a honeycomb. It is fairly easy to see why a low-level filter might detect edges. Consider a situation in which the color of the image changes along an edge. As a result, the difference between neighboring pixel values will be non-zero only across the edge. This can be achieved by choosing the appropriate weights in the corresponding low-level filter. Note that the filter to detect a horizontal edge will not be the same as that to detect a vertical edge. This brings us back to Hubel and Weisel's experiments in which different neurons in the cat's visual cortex were activated by different edges. Examples of filters detecting horizontal and vertical edges are illustrated in Figure 9.6. The next layer filter works on the hidden features and therefore it is harder to interpret. Nevertheless, the next layer filter is able to detect a rectangle by combining the horizontal and vertical edges.

In a later section, we will show visualizations of how smaller portions of real-world image activate different hidden features, much like the biological model of Hubel and Wiesel in which different shapes seem to activate different neurons. Therefore, the power of convolutional neural networks rests in the ability to put together these primitive shapes into more complex shapes layer by layer. Note that it is impossible for the first convolution layer to learn any feature that is larger than $F_1 \times F_1$ pixels, where the value of F_1 is typically a small

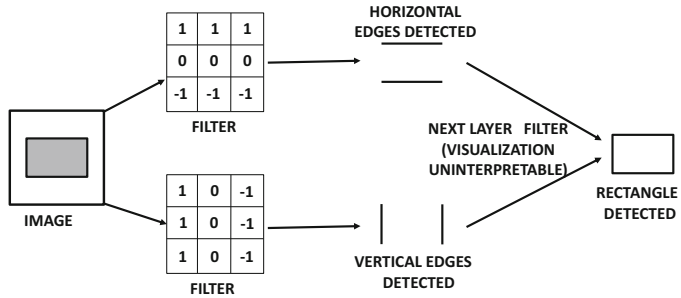


Figure 9.6: Filters detect edges and combine them to create rectangle.

number like 3 or 5. However, the next convolution layer will be able to put together many of these patches together to create a feature from an area of the image that is larger. The primitive features learned in earlier layers are put together in a semantically coherent way to learn increasingly complex and interpretable visual features. The choice of learned features is affected by how backpropagation adapts the features to the needs of the loss function at hand. For example, if an application is training to classify images as cars, the approach might learn to put together arcs to create a circle, and then it might put together circles with other shapes to create a car wheel. All this is enabled by the hierarchical features of a deep network.

Recent *ImageNet* competitions have demonstrated that much of the power in image recognition lies in increased depth of the network. Not having enough layers effectively prevents the network from learning the hierarchical regularities in the image that are combined to create its semantically relevant components. Another important observation is that the nature of the features learned will be sensitive to the specific data set at hand. For example, the features learned to recognize trucks will be different from those learned to recognize carrots. However, some data sets (like *ImageNet*) are diverse enough that the features learned by training on these data sets have general-purpose significance across many applications.

9.3 Training a Convolutional Network

The process of training a convolutional neural network uses the backpropagation algorithm. There are primarily three types of layers, corresponding to the convolution, ReLU, and max-pooling layers. We will separately describe the backpropagation algorithm through each of these layers. The ReLU is relatively straightforward to backpropagate through because it is no different than a traditional neural network. For max-pooling with no overlap between pools, one only needs to identify which unit is the maximum value in a pool (with ties broken arbitrarily or divided proportionally). The partial derivative of the loss with respect to the pooled state flows back to the unit with maximum value. All entries other than the maximum entry in the grid will be assigned a value of 0. Note that the backpropagation through a maximization operation is also described in Table 2.1 of Chapter 2. For cases in which the pools are overlapping, let $P_1 \ldots P_r$ be the pools in which the unit h is involved, with corresponding activations $h_1 \ldots h_r$ in the next layer. If h is the maximum value in pool P_i (and therefore $h_i = h$), then the gradient of the loss with respect to h_i flows back to h (with ties broken arbitrarily or divided proportionally). The contributions of the different overlapping pools (from $h_1 \ldots h_r$ in the next layer) are added in order to compute the gradient with respect to the unit h. Therefore, the backpropagation through the maximization and the ReLU operations are not very different from those in traditional neural networks.

9.3.1 Backpropagating Through Convolutions

The backpropagation through convolutions is also not very different from the backpropagation with linear transformations (i.e., matrix multiplications) in a feed-forward network. This point of view will become particularly clear when we present convolutions as a form of matrix multiplication. Just as backpropagation in feed-forward networks from layer $(i+1)$ to layer i is achieved by multiplying the error derivatives with respect to layer $(i+1)$ with the transpose of the forward propagation matrix between layers i and $(i+1)$ (cf. Table 2.1 of Chapter 2), backpropagation in convolutional networks can also be seen as a form of transposed convolution.

First, we describe a simple element-wise approach to backpropagation. Assume that the loss gradients of the cells in layer $(i+1)$ have already been computed. The loss derivative with respect to a cell in layer $(i+1)$ is defined as the partial derivative of the loss function with respect to the hidden variable in that cell. Convolutions multiply the activations in layer i with filter elements to create elements in the next layer. Therefore, a cell in layer $(i+1)$ receives aggregated contributions from a 3-dimensional volume of elements in the previous layer of filter size $F_i \times F_i \times d_i$. At the same time, a cell c in layer i contributes to multiple elements (denoted by set S_c) in layer $(i+1)$, although the number of elements to which it contributes depends on the depth of the next layer and the stride. Identifying this "forward set" is the key to the backpropagation. A key point is that the cell c contributes to each element in S_c in an additive way after multiplying the activation of cell c with a filter element. Therefore, backpropagation simply needs to multiply the loss derivative of each element in S_c with respect to the corresponding filter element and aggregate in the backwards direction at c. For any particular cell c in layer i, the following pseudocode can be used to backpropagate the existing derivatives in layer-$(i+1)$ to cell c in layer-i:

Identify all cells S_c in layer $(i+1)$ to which cell c in layer i contributes;
For each cell $r \in S_c$, let δ_r be its (already backpropagated) loss-derivative with respect to cell r;
For each cell $r \in S_c$, let w_r be weight of filter element used for contributing from cell c to r;
$\delta_c = \sum_{r \in S_c} \delta_r \cdot w_r$;

After the loss gradients have been computed, the values are multiplied with those of the hidden units of the $(i-1)$th layer to obtain the gradients with respect to the weights between the $(i-1)$th and ith layer. In other words, the hidden value at one end point of a weight is multiplied with the loss gradient at the other end in order to obtain the partial derivative with respect to the weight. However, this computation assumes that all weights are distinct, whereas the weights in the filter are shared across the entire spatial extent of the layer. Therefore, one has to be careful to account for shared weights, and sum up the partial derivatives of all copies of a shared weight. In other words, we first pretend that the filter used in each position is distinct in order to compute the partial derivative with respect to each copy of the shared weight, and then add up the partial derivatives of the loss with respect to all copies of a particular weight.

Note that the approach above uses simple linear accumulation of gradients like traditional backpropagation. However, one has to be slightly careful in terms of keeping track of the cells that influence other cells in the next layer. One can implement backpropagation with the help of tensor multiplication operations, which can further be simplified into simple matrix multiplications of derived matrices from these tensors. This point of view will be discussed in the next two sections because it provides many insights on how many aspects of feedforward networks can be generalized to convolutional neural networks.

9.3.2 Backpropagation as Convolution with Inverted/Transposed Filter

In conventional neural networks, a backpropagation operation is performed by multiplying a vector of gradients at layer $(q+1)$ with the transposed weight matrix between the layers q and $(q+1)$ in order to obtain the vector of gradients at layer q (cf. Table 2.1). In convolution neural networks, the backpropagated derivatives are also associated with spatial positions in the layers. Is there an analogous convolution we can apply to the spatial footprint of backpropagated derivatives in a layer to obtain those of the previous layer? It turns out that this is indeed possible.

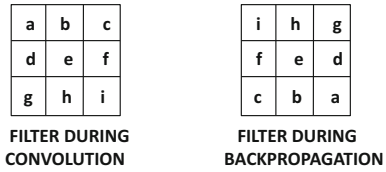


Figure 9.7: The inverse of a kernel for backpropagation

Let us consider the case in which the activations in layer q are convolved with a filter to create those in layer $(q+1)$. For simplicity, consider the case in which depth d_q of the input layer and the depth d_{q+1} of the output layer are both 1; furthermore, we use convolutions with stride 1. In such a case, the convolution filter is inverted both horizontally and vertically for backpropagation. An example of such an inverted filter is illustrated in Figure 9.7. The intuitive reason for this inversion is that the *filter* is "moved around" the spatial area of the input volume to perform the dot product, whereas the backpropagated derivatives are with respect to the *input volume*, whose relative movement with respect to the filter is the opposite of the filter movement during convolutions. Note that the entry in the extreme upper-left of the convolution filter might not even contribute to the extreme upper-left entry in the output volume (because of padding), but it will almost always contribute to the extreme lower-right entry of the output volume. This is consistent with the inversion of the filter. The backpropagated derivative set of the $(q+1)$th layer is convolved with this inverted filter to obtain the backpropagated derivative set of the qth layer. How are the paddings of the forward convolution and backward convolution related? For a stride of 1, the sum of the paddings during forward propagation and backward propagation is $F_q - 1$, where F_q is the side length of the filter for qth layer.

Now consider the case in which the depths d_q and d_{q+1} are no longer 1, but are arbitrary values. In this case, an additional tensor transposition needs to occur. The weight of the (i, j, k)th position of the pth filter in the qth layer is $\mathcal{W} = [w_{ijk}^{(p,q)}]$. Note that i and j refer to spatial positions, whereas k refers to the depth-centric position of the weight. In such a case, let the 5-dimensional tensor corresponding to the backpropagation filters from layer $q+1$ to layer q be denoted by $\mathcal{U} = [u_{ijk}^{(p,q+1)}]$. Then, the entries of this tensor are as follows:

$$u_{rsp}^{(k,q+1)} = w_{ijk}^{(p,q)} \tag{9.1}$$

Here, we have $r = F_q - i + 1$ and $s = F_q - j + 1$. Note that the index p of the filter identifier and depth k within a filter have been interchanged between $\mathcal{W}$ and $\mathcal{U}$ in Equation 9.1. This is a tensor-centric transposition.

In order to understand the transposition above, consider a situation in which we use 20 filters on the 3-channel RGB volume in order to create an output volume of depth 20. While backpropagating, we will need to take a *gradient volume* of depth 20 and transform to a *gradient volume* of depth 3. Therefore, we need to create 3 filters for backpropagation, each of which is for the red, green, and blue colors. We pull out the 20 spatial slices from the 20 filters that are applied to the red color, invert them using the approach of Figure 9.7, and then create a single 20-depth filter for backpropagating gradients with respect to the red slice. Similar approaches are used for the green and blue slices. The transposition and inversion in Equation 9.1 correspond to these operations.

9.3.3 Convolution/Backpropagation as Matrix Multiplications

It is helpful to view convolution as a matrix multiplication because it helps us define various related notions such as *transposed convolution*, *deconvolution*, and *fractional convolution*. These concepts are helpful not just in understanding backpropagation, but also in developing the machinery necessary for convolutional autoencoders. In traditional feed-forward networks, matrices that are used to transform hidden states in the forward phase are transposed in the backwards phase (cf. Table 2.1) in order to backpropagate partial derivatives across layers. Similarly, the matrices used in encoders are often transposed in the decoders when working with autoencoders in traditional settings. Although the spatial structure of the convolutional neural network does mask the nature of the underlying matrix multiplication, one can "flatten" this spatial structure to perform the multiplication and reshape back to a spatial structure using the known spatial positions of the elements of the flattened matrix. This somewhat indirect approach is helpful in understanding the fact that the convolution operation is similar to the matrix multiplication in feed-forward networks at a very fundamental level. Furthermore, real-world implementations of convolution are often accomplished with matrix multiplication.

For simplicity, let us first consider the case in which the qth layer and the corresponding filter used for convolution both have unit depth. Furthermore, assume that we are using a stride of 1 with zero padding. Therefore, the input dimensions are $L_q \times B_q \times 1$, and the output dimensions are $(L_q - F_q + 1) \times (B_q - F_q + 1) \times 1$. In the common setting in which the spatial dimensions are square (i.e., $L_q = B_q$), one can assume that the spatial dimensions of the input $A_I = L_q \times L_q$ and the spatial dimensions of the output are $A_O = (L_q - F_q + 1) \times (L_q - F_q + 1)$. Here, A_I and A_O are the spatial areas of the input and output matrices, respectively. The input can be representing by flatting the area A_I into an A_I-dimensional column vector in which the rows of the spatial area are concatenated from top to bottom. This vector is denoted by $\overline{f}$. An example of a case in which we use a 2×2 filter on a 3×3 input is shown in Figure 9.8. Therefore, the output is of size 2×2, and we have $A_I = 3 \times 3 = 9$, and $A_O = 2 \times 2 = 4$. The 9-dimensional column vector for the 3×3 input is shown in Figure 9.8. A sparse matrix C is defined in lieu of the filter, which is the key in representing the convolution as a matrix multiplication. A matrix of size $A_O \times A_I$ is defined in which each row corresponds to the convolution at one of the A_O convolution locations. These rows are associated with the spatial location of the top-left corner of the convolution region in the input matrix from which they were derived. The value of each entry in the row corresponds to one of the A_I positions in the input matrix, but this value is 0, if that input position is not involved in the convolution for that row. Otherwise, the value is set to the corresponding value of the filter, which is used for multiplication. The ordering of the entries in a row is based on the same spatially sensitive ordering of the input matrix locations as was used to flatten the input matrix into an A_I-dimensional vector. Since the filter size is usually much smaller than the input size, most of the entries in the matrix C are 0s, and each entry of the filter occurs in every row of C. Therefore, every entry in the filter is repeated A_O times in C, because it is used for A_O multiplications.

An example of a 4×9 matrix C is shown in Figure 9.8. Subsequent multiplication of C with $\overline{f}$ yields an A_O-dimensional vector. The corresponding 4-dimensional vector is shown in Figure 9.8. Since each of the A_O rows of C is associated with a spatial location, these locations are inherited by $C\overline{f}$. These spatial locations are used to reshape $C\overline{f}$ to a spatial matrix. The reshaping of the 4-dimensional vector to a 2×2 matrix is also shown in Figure 9.8.

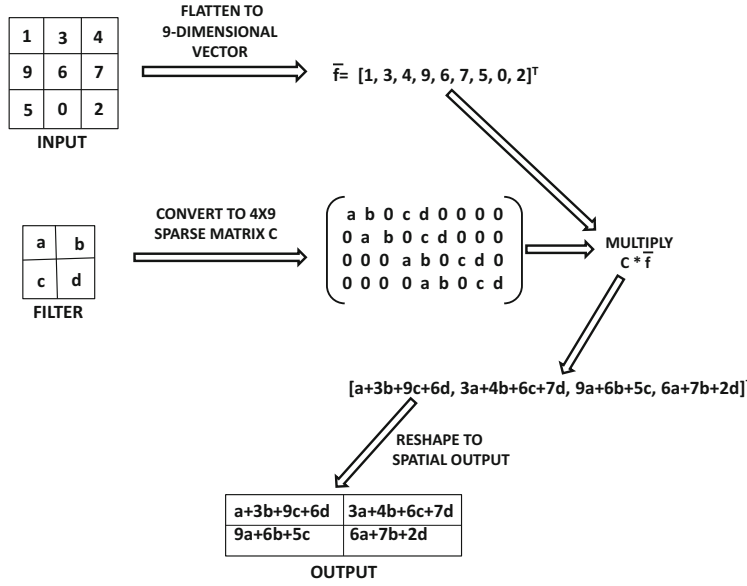


Figure 9.8: Convolution as matrix multiplication

This particular exposition uses the simplified case with a depth of 1. In the event that the depth is larger than 1, the same approach is applied for each 2-dimensional slice, and the results are added. In other words, we aggregate $\sum_p C_p \overline{f}_p$ over the various slice indices p and then the results are re-shaped into a 2-dimensional matrix. This approach amounts to a *tensor* multiplication, which is a straightforward generalization of a matrix multiplication. The tensor multiplication approach is how convolution is actually implemented in practice. In general, one will have multiple filters, which correspond to multiple output maps. In such a case, the kth filter will be converted into the sparsified matrix $C_{p,k}$, and the kth feature map of the output volume will be $\sum_p C_{p,k} \overline{f}_p$.

The matrix-centric approach is very useful for performing backpropagation because one can also propagate gradients backwards by using the same approach in the backwards direction, except that the *transposed* matrix C^T is used for multiplication with the flattened vector version of a 2-dimensional slice of the output gradient. Note that the flattening of a gradient with respect to a spatial map can be done in a similar way as the flattened vector $\overline{f}$ is created in the forward phase. Consider the simple case in which both the input and output volumes have a depth of 1. If $\overline{g}$ is the flattened vector gradient of the loss with respect to the output spatial map, then the flattened gradient with respect to the input spatial map is obtained as $C^T \overline{g}$. This approach is consistent with the approach used in feed-forward networks, in which the transpose of the forward matrix is used in backpropagation. The above result is for the simple case when both input and output volumes have depth of 1. What happens in the general case? When the depth of the output volume is $d > 1$, the gradients with respect to the output maps are denoted by $\overline{g}_1 \ldots \overline{g}_d$. The corresponding gradient with respect to the features in the pth spatial slice of the input volume is given by $\sum_{k=1}^{d} C_{p,k}^T \overline{g}_k$. Here, the matrix $C_{p,k}$ is obtained by converting the pth spatial slice of the kth filter into the sparsified matrix as discussed above. This approach is a consequence of Equation 9.1. This type of transposed convolution is also useful for the deconvolution operation in convolution autoencoders, which will be discussed later in this chapter (cf. section 9.5).

9.3.4 Data Augmentation

A common trick to reduce overfitting in convolutional neural networks is the idea of *data augmentation*. In data augmentation, new training examples are generated by using transformations on the original examples. Data augmentation is very well suited to the image setting because many transformations such as translation, rotation, patch extraction, and reflection do not fundamentally change the properties of the object in an image. Augmenting the training data with these transformations increases the generalization power of the model. For example, if a data set is trained with mirror images and reflected versions of all the bananas in it, then the model is able to better recognize bananas in different orientations.

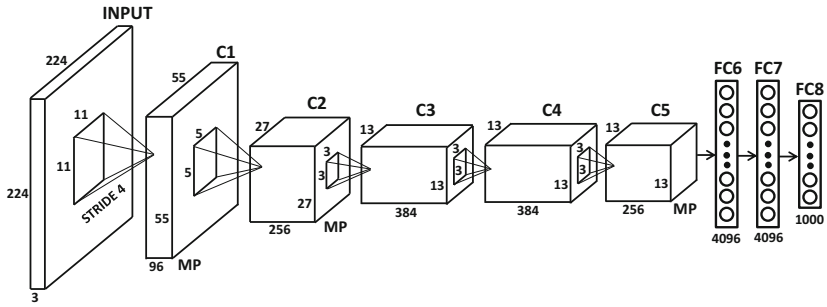
Many of these forms of data augmentation require very little computation, and therefore the augmented images do not need to be explicitly generated up front. Rather, they can be created at training time, when an image is being processed. For example, while processing an image of a banana, it can be reflected into a modified banana at training time. Similarly, the same banana might be represented in somewhat different color intensities in different images, and therefore it might be helpful to create representations of the same image in different color intensities. In many cases, creating the training data set using image patches can be helpful. An important neural network that rekindled interest in deep learning by winning the ILSVRC challenge was *AlexNet*. This network was trained by extracting $224 \times 224 \times 3$ patches from the images, which also defined the input sizes for the networks. The neural networks, which were entered into the ILSVRC contest in subsequent years, used a similar methodology of extracting patches.

Although most data augmentation methods are quite efficient, some forms of transformation that use principal component analysis (PCA) can be more expensive. PCA is used in order to change the color intensity of an image. If the computational costs are high, it becomes important to extract the images up front and store them. The basic idea here is to use the 3×3 covariance matrix of each pixel value and compute the principal components. Then, Gaussian noise is added to each principal component with zero mean and variance of 0.01. This noise is fixed over all the pixels of a particular image. The approach is dependent on the fact that object identity is invariant to color intensity and illumination. It is reported in [263] that data set augmentation reduces error rate by 1%.

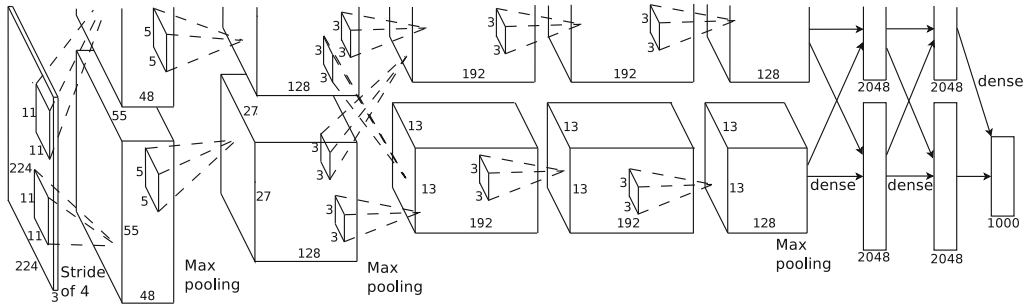
One must be careful not to apply data augmentation blindly without regard to the data set and application at hand. For example, applying rotations and reflections on the MNIST data set [287] of handwritten digits is a bad idea because the digits in the data set are all presented in a similar orientation. Furthermore, the mirror image of an asymmetric digit is not a valid digit, and a rotation of a '6' is a '9.' The key point in deciding what types of data augmentation are reasonable is to account for the natural distribution of images in the full data set, as well as the effect of a specific type of transformation on the class labels.

9.4 Case Studies of Convolutional Architectures

In the following, we provide some case studies of convolutional architectures. These case studies were derived from successful entries to the ILSVRC competition in recent years. These are instructive because they provide an understanding of the important factors in neural network design that can make these networks work well. Even though recent years have seen some changes in architectural design (like ReLU activation), it is striking how similar the modern architectures are to the basic design of *LeNet-5*. The main changes from *LeNet-5* to modern architectures are in terms of the explosion of depth, the use of ReLU activation, and the training efficiency enabled by modern hardware/optimization en-



(a) Without GPU partitioning



(b) With GPU partitioning (original architecture)

Figure 9.9: The *AlexNet* architecture. The ReLU activations follow each convolution layer, and are not explicitly shown. Note that the max-pooling layers are labeled as MP, and they follow only a subset of the convolution-ReLU combination layers. The architectural diagram in (b) is from [A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS Conference*, pp. 1097–1105. 2012.] ©2012 A. Krizhevsky, I. Sutskever, and G. Hinton.

hancements. Modern architectures are deeper, and they use a variety of computational, architectural, and hardware tricks to efficiently train these networks with large amounts of data. Hardware advancements should not be underestimated; modern GPU-based platforms are 10,000 times faster than the (similarly priced) systems available at the time *LeNet-5* was proposed. Even on these modern platforms, it often takes a week to train a convolutional neural network that is accurate enough to be competitive at ILSVRC. The hardware, data-centric, and algorithmic enhancements are connected to some extent. It is difficult to try new algorithmic tricks if enough data and computational power is not available to experiment with complex/deeper models in a reasonable amount of time. Therefore, the recent revolution in deep convolutional networks could not have been possible, had it not been for the large amounts of data and increased computational power available today. In the following sections, we provide an overview of some of the well-known models for image classification.

9.4.1 AlexNet

AlexNet was the winner of the 2012 ILSVRC competition. The architecture of *AlexNet* is shown in Figure 9.9(a). It is worth mentioning that there were two parallel pipelines of

processing in the original architecture, which are not shown in Figure 9.9(a). These two pipelines are caused by two GPUs working together to build the training model more efficiently. The network was originally trained on a GTX 580 GPU with 3 GB of memory, and it was impossible to fit the intermediate computations in this amount of space. Therefore, the network was partitioned across two GPUs. The original architecture is shown in Figure 9.9(b), in which the work is partitioned into two GPUs. We also show the architecture without the changes caused by the GPUs, so that it can be more easily compared with other convolutional neural network architectures discussed in this chapter. It is noteworthy that the GPUs are inter-connected in only a subset of the layers in Figure 9.9(b), which leads to some differences between Figure 9.9(a) and 9.9(b) in terms of the actual model constructed. Specifically, the GPU-partitioned architecture has fewer weights because not all layers have interconnections. Dropping some of the interconnections reduces the communication time between the processors and therefore helps in efficiency.

AlexNet starts with $224 \times 224 \times 3$ images and uses 96 filters of size $11 \times 11 \times 3$ in the first layer. A stride of 4 is used. This results in a first layer of size $55 \times 55 \times 96$. After the first layer has been computed, a max-pooling layer is used. This layer is denoted by 'MP' in Figure 9.9(a). Note that the architecture of Figure 9.9(a) is a simplified version of the architecture shown in Figure 9.9(b), which explicitly shows the two parallel pipelines. For example, Figure 9.9(b) shows a depth of the first convolution layer of only 48, because the 96 feature maps are divided among the GPUs for parallelization. On the other hand, Figure 9.9(a) does not assume the use of GPUs, and therefore the width is explicitly shown as 96. The ReLU activation function was applied after each convolutional layer, which was followed by response normalization and max-pooling. Although max-pooling has been annotated in the figure, it has not been assigned a block in the architecture. Furthermore, the ReLU and response normalization layers are not explicitly shown in the figure. These types of concise representations are common in pictorial depictions of neural architectures.

The second convolutional layer uses the response-normalized and pooled output of the first convolutional layer and filters it with 256 filters of size $5 \times 5 \times 96$. No intervening pooling or normalization layers are present in the third, fourth, or fifth convolutional layers. The sizes of the filters of the third, fourth, and fifth convolutional layers are $3 \times 3 \times 256$ (with 384 filters), $3 \times 3 \times 384$ (with 384 filters), and $3 \times 3 \times 384$ (with 256 filters). All max-pooling layers used 3×3 filters at stride 2. Therefore, there was some overlap among the pools. The fully connected layers have 4096 neurons. The final set of 4096 activations can be treated as a 4096-dimensional representation of the image. The final layer of *AlexNet* uses a 1000-way softmax in order to perform the classification. It is noteworthy that the final layer of 4096 activations (labeled by FC7 in Figure 9.9(b)) is often used to create a flat 4096 dimensional representation of an image for applications beyond classification. One can extract these features for any out-of-sample image by simply passing it through the trained neural network. These features often generalize well to other data sets and other tasks. Such features are referred to as FC7 features. In fact, the use of the extracted features from the penultimate layer as FC7 was popularized after *AlexNet*, even though the approach was known much earlier. As a result, such extracted features from the penultimate layer of a convolutional neural network are often referred to as *FC7 features*, irrespective of the number of layers in that network. It is noteworthy that the number of feature maps in middle layers is far larger than the initial depth of the volume in the input layer (which is only 3 corresponding to RGB colors) although their spatial dimensions are smaller. This is because the initial depth only contains the RGB color components, whereas the later layers capture different types of semantic features in the features maps.

Many design choices used in the architecture became standard in later architectures. A specific example is the use of ReLU activation in the architecture (instead of sigmoid or tanh units). The use of the ReLU has now become a default choice, although this was not the case before *AlexNet*. Some other training tricks were popularized by *AlexNet* (although they were known earlier). One example was the use of data augmentation, which turned out to be very useful in improving accuracy. *AlexNet* also underlined the importance of using specialized hardware like GPUs for training on such large data sets. Dropout was used with L_2-weight decay in order to improve generalization. The use of Dropout is common in virtually all types of architectures today because it provides an additional booster in most cases. *AlexNet* also used a normalization within the layers, referred to as *local response normalization*; however, this trick was later found to be ineffective, and its use was eventually discontinued.

We also briefly mention the parameter choices used in *AlexNet*. The interested reader can find the full code and parameter files of *AlexNet* at [608]. L_2-regularization was used with a parameter of 5×10^{-4}. *Dropout* was used by sampling units at a probability of 0.5. Momentum-based (mini-batch) stochastic gradient descent was used for training *AlexNet* with parameter value of 0.8. The batch-size was 128. The learning rate was 0.01, although it was eventually reduced a couple of times as the method began to converge. Even with the use of the GPU, the training time of *AlexNet* was of the order of a week.

The final top-5 error rate, which was defined as the fraction of cases in which the correct image was not included in the top-5 images, was about 15.4%. This error rate[2] was in comparison with the previous winners with an error rate of more than 25%. The gap with respect to the second-best performer in the contest was also similar. The use of single convolutional network provided a top-5 error rate of 18.2%, although using an ensemble of seven models provided the winning error-rate of 15.4%. Note that these types of ensemble-based tricks provide a consistent improvement of between 2% and 3% with most architectures. Furthermore, since the executions of most ensemble methods are embarrassingly parallelizable, it is relatively easy to perform them, as long as sufficient hardware resources are available. *AlexNet* is considered a fundamental advancement within the field of computer vision because of the large margin with which it won the ILSVRC contest. This success rekindled interest in deep learning in general, and convolutional neural networks in particular.

9.4.2 ZFNet

A variant of *ZFNet* [581] was the winner of the ILSVRC competition in 2013. Its architecture was heavily based on *AlexNet*, although some changes were made to further improve the accuracy. Most of these changes were associated with differences in hyperparameter choices, and therefore *ZFNet* is not very different from *AlexNet* at a fundamental level. One change from *AlexNet* to *ZFNet* was that the initial filters of size $11 \times 11 \times 3$ were changed to $7 \times 7 \times 3$. Instead of strides of 4, strides of 2 were used. The second layer used 5×5 filters at stride 2 as well. As in *AlexNet*, there are three max-pooling layers, and the same sizes of max-pooling filters were used. However, the first pair of max-pooling layers were performed after the first and second convolutions (rather than the second and third convolutions). As a result, the spatial footprint of the third layer changed to 13×13 rather than 27×27, although all other spatial footprints remained unchanged from *AlexNet*. The sizes of various layers in *AlexNet* and *ZFNet* are listed in Table 9.1.

The third, fourth, and fifth convolutional layers use a larger number of filters in *ZFNet* as compared to *AlexNet*. The number of filters in these layers were changed from $(384, 384, 256)$ to $(512, 1024, 512)$. As a result, the spatial footprints of *AlexNet* and *ZFNet* are the same

[2]The top-5 error rate makes more sense in image data where a single image might contain objects of multiple classes. Throughout this chapter, we use the term "error rate" to refer to the top-5 error rate.

Table 9.1: Comparison of *AlexNet* and *ZFNet*

	AlexNet	*ZFNet*
Volume: Operations:	$224 \times 224 \times 3$ Conv 11×11 (stride 4)	$224 \times 224 \times 3$ Conv 7×7 (stride 2), MP
Volume: Operations:	$55 \times 55 \times 96$ Conv 5×5, MP	$55 \times 55 \times 96$ Conv 5×5 (stride 2), MP
Volume: Operations:	$27 \times 27 \times 256$ Conv 3×3, MP	$13 \times 13 \times 256$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 512$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 1024$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 256$ MP, Fully connect	$13 \times 13 \times 512$ MP, Fully connect
FC6: Operations:	4096 Fully connect	4096 Fully connect
FC7: Operations:	4096 Fully connect	4096 Fully connect
FC8: Operations:	1000 Softmax	1000 Softmax

in most layers, although the depths are different in the final three convolutional layers with similar spatial footprints. From an overall perspective, *ZFNet* used similar principles to *AlexNet*, and the main gains were obtained by changing the architectural parameters of *AlexNet*. This architecture reduced the top-5 error rate to 14.8% from 15.4%, and further increases in width/depth from the same author(s) reduced the error to 11.1%. Since most of the differences between *AlexNet* and *ZFNet* were those of minor design choices, this emphasizes the fact that small details are important when working with deep learning algorithms. Thus, extensive experimentation with neural architectures are sometimes important in order to obtain the best performance. The architecture of *ZfNet* was made wider and deeper, and the results were submitted to ILSVRC in 2013 under the name *Clarifai*, which was a company[3] founded by the first author of [581]. The difference[4] between *Clarifai* and *ZFNet* was one of width/depth of the network, although exact details of these differences are not available. This entry was the winning entry of the ILSVRC competition in 2013. Refer to [581] for details and a pictorial illustration of the architecture.

9.4.3 VGG

VGG [474] further emphasized the developing trend in terms of increased depth of networks. The tested networks were designed with various configurations with sizes between 11 and 19 layers, although the best-performing versions had 16 or more layers. *VGG* was a top-performing entry on ISLVRC in 2014, but it was not the winner. The winner was *GoogLeNet*, which had a top-5 error rate of 6.7% in comparison with the top-5 error rate of 7.3% for *VGG*. Nevertheless, *VGG* was important because it illustrated several important design principles that eventually became standard in future architectures.

An important innovation of *VGG* is that it reduced filter sizes but increased depth. It is important to understand that *reduced filter size necessitates increased depth*. This is

[3] http://www.clarifai.com

[4] Personal communication from Matthew Zeiler.

because a small filter can capture only a small region of the image unless the network is deep. For example, a single feature that is a result of three sequential convolutions of size 3×3 will capture a region in the input of size 7×7. Note that using a single 7×7 filter directly on the input data will also capture the visual properties of a 7×7 input region. In the first case, we are using $3 \times 3 \times 3 = 27$ parameters, whereas we are using $7 \times 7 \times 1 = 49$ parameters in the second case. Therefore, the parameter footprint is smaller in the case when three sequential convolutions are used. However, three successive convolutions can often capture more interesting and complex features than a single convolution, and the resulting activations with a single convolution will look like primitive edge features. Therefore, the network with 7×7 filters will be unable to capture sophisticated shapes in smaller regions.

A deeper network will have more nonlinearity in feature engineering because of the presence of more ReLU layers, and more regularization because the increased depth forces a structure on the layers through the use of repeated composition of convolutions. As discussed above, architectures with greater depth and reduced filter size require fewer parameters. This occurs in part because the number of parameters in each layer is given by the square of the filter size, whereas the number of parameters depend linearly on the depth. Therefore, one can drastically reduce the number of parameters by using smaller filter sizes, and instead "spend" these parameters by using increased depth. Increased depth improves feature engineering and discriminative power. Therefore *VGG* always uses filters with spatial footprint 3×3 and pooling of size 2×2. The convolution was done with stride 1, and a padding of 1 was used. The pooling was done at stride 2. Using a 3×3 filter with a padding of 1 maintains the spatial footprint of the output volume, although pooling always compresses the spatial footprint. Therefore, the pooling was done on non-overlapping spatial regions (unlike the previous two architectures), and always reduced the spatial footprint (i.e., both height and width) by a factor of 2. Another interesting design choice of *VGG* was that the number of filters was often increased by a factor of 2 after each max-pooling. The idea was to always increase the depth by a factor of 2 whenever the spatial footprint reduced by a factor of 2. This design choice results in some level of balance in the computational effort across layers, and was inherited by some of the later architectures like *ResNet*.

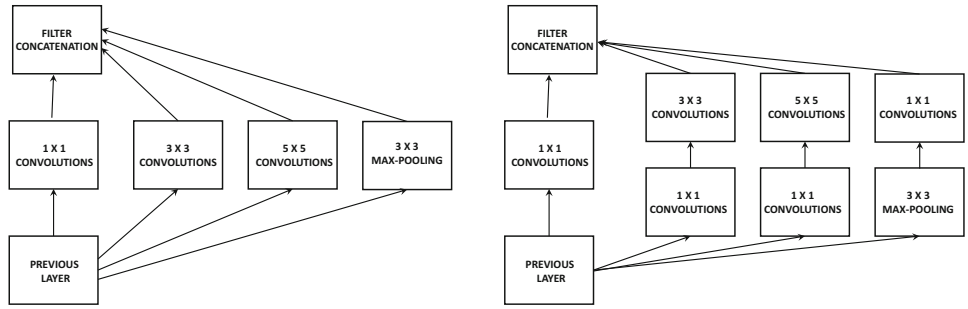
One issue with using deep configurations was that increased depth led to greater sensitivity with initialization, which is known to cause instability. This problem was solved by using pretraining, in which a shallower architecture was first trained, and then further layers were added. However, the pretraining was not done on a layer-by-layer basis. Rather, an 11-layer subset of the architecture was first trained. These trained layers were used to initialize a subset of the layers in the deeper architecture. *VGG* achieved a top-5 error of only 7.3% in the ISLVRC contest, which was one of the top performers but not the winner. The different configurations of *VGG* are shown in Table 9.2. Among these, the architecture denoted by column D was the winning architecture. Note that the number of filters increase by a factor of 2 after each max-pooling. Therefore, max-pooling causes the spatial height and width to reduce by a factor of 2, but this is compensated by increasing depth by a factor of 2. Performing convolutions with 3×3 filters and padding of 1 does not change the spatial footprint. Therefore, the sizes of each spatial dimension (i.e., height and width) in the regions between different max-pooling layers in column D of Table 9.2 are 224, 112, 56, 28, and 14, respectively. A final max-pooling is performed just before creating the fully connected layer, which reduces the spatial footprint further to 7. Therefore, the first fully connected layer has dense connections between 4096 neurons and a $7 \times 7 \times 512$ volume. As we will see later, most of the parameters of the neural network are hidden in these connections.

An interesting exercise has been shown in [246] about where most of the parameters and the memory of the activations is located. In particular, the vast majority of the *memory*

Table 9.2: Configurations used in *VGG*. The term C3D64 refers to the case in which convolutions are performed with 64 filters of spatial size 3×3 (and occasionally 1×1). The depth of the filter matches the corresponding layer. The padding of each filter is chosen in order to maintain the spatial footprint of the layer. All convolutions are followed by ReLU. The max-pool layer is referred to as M, and local response normalization as LRN. The softmax layer is denoted by S, and FC4096 refers to a fully connected layer with 4096 units. Other than the final set of layers, the number of filters always increases after each max-pooling. Therefore, reduced spatial footprint is often accompanied with increased depth.

Name:	**A**	**A-LRN**	**B**	**C**	**D**	**E**
# Layers	11	11	13	16	16	19
	C3D64	C3D64	C3D64	C3D64	C3D64	C3D64
		LRN	C3D64	C3D64	C3D64	C3D64
	M	M	M	M	M	M
	C3D128	C3D128	C3D128	C3D128	C3D128	C3D128
			C3D128	C3D128	C3D128	C3D128
	M	M	M	M	M	M
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
				C1D256	C3D256	C3D256
						C3D256
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC1000	FC1000	FC1000	FC1000	FC1000	FC1000
	S	S	S	S	S	S

required for storing the activations and gradients in the forward and backward phases are required by the early part of the convolutional neural network with the largest spatial footprint. This point is significant because the memory required by a mini-batch is scaled by the size of the mini-batch. For example, it is has been shown in [246] that about 93MB are required for each image. Therefore, for a mini-batch size of 128, the total memory requirement would be about 12GB. Although the early layers require the most memory because of their large spatial footprints, they do not have a large parameter footprint because of the sparse connectivity and weight sharing. In fact, most of the parameters are required by the fully connected layers at the end. The connection of the final $7 \times 7 \times 512$ spatial layer (cf. column D in Table 9.2) to the 4096 neurons required $7 \times 7 \times 512 \times 4096 = 102,760,448$ parameters. The total number of parameters in *all* layers was about $138,000,000$. Therefore, *nearly* 75%



(a) Basic inception module (b) Implementation with 1×1 bottlenecks

Figure 9.10: The inception module of *GoogLeNet*

of the parameters are in a single layer of connections. Furthermore, the majority of the remaining parameters are in the final two fully connected layers. In all, dense connectivity accounts for 90% of the parameter footprint in the neural network. This point is significant, as *GoogLeNet* uses some innovations to reduce the parameter footprint in the final layers.

It is notable that some of the architectures allow 1×1 convolutions. Although a 1×1 convolution does not combine the activations of spatially adjacent features, it does combine the feature values of different channels when the depth of a volume is greater than 1. Using a 1×1 convolution is also a way to incorporate additional nonlinearity into the architecture without making fundamental changes at the spatial level. This additional nonlinearity is incorporated via the ReLU activations attached to each layer. Refer to [474] for more details.

9.4.4 GoogLeNet

GoogLeNet proposed a novel concept referred to as an *inception architecture*. An inception architecture is a *network within a network*. The initial part of the architecture is much like a traditional convolutional network, and is referred to as the *stem*. The key part of the network is an intermediate layer, referred to as an *inception module*. An example of an inception module is illustrated in Figure 9.10(a). The basic idea of the inception module is that key information in the images is available at different levels of detail. If we use a large filter, we can capture information in a bigger area containing limited variation; if we use a smaller filter, we can capture detailed information in a smaller area. While one solution would be to pipe together many small filters, this would be wasteful of parameters and depth when it would suffice to use the broader patterns in a larger area. The problem is that we do not know up front which level of detail is appropriate for each region of the image. Why not give the neural network the flexibility to model the image at different levels of granularities? This is achieved with an inception module, which convolves with three different filter sizes in parallel. These filter sizes are 1×1, 3×3, and 5×5. A purely sequential piping of filters of the same size is inefficient when one is faced with objects of different scales in different images. Since all filters on the inception layer are learnable, the neural network can decide which ones will influence the output the most. By choosing filters of different sizes along different paths, different regions are represented at a different level of granularity. *GoogLeNet* is made up of nine inception modules that are arranged sequentially. Therefore, one can choose many alternative paths through the architecture, and the resulting features will represent very different spatial regions. For example, passing through four 3×3 filters

followed by only 1×1 filters will capture a relatively small spatial area. On the other hand, passing through many 5×5 filters will result in a much larger spatial footprint. In other words, the differences in the scales of the shapes captured in different hidden features will be magnified in later layers. In recent years, batch normalization has been used in conjunction with the inception architecture, which simplifies[5] the network structure from its original form.

One observation is that the inception module results in some computational inefficiency because of the large number of convolutions of different sizes. Therefore, an efficient implementation is shown in Figure 9.10(b), in which 1×1 convolutions are used to first reduce the depth of the feature map. This is because the number of 1×1 convolution filters is a modest factor less than the depth of the input volume. For example, one might first reduce an input depth of 256 to 64 by using 64 different 1×1 filters. These additional 1×1 convolutions are referred to as the *bottleneck operations* of the inception module. Initially reducing the depth of the feature map (with cheap 1×1 convolutions) saves computational efficiency with the larger convolutions because of the reduced depth of the layers after applying the bottleneck convolutions. One can view the 1×1 convolutions as a kind of supervised dimensionality reduction before applying the larger spatial filters. The dimensionality reduction is supervised because the parameters in the bottleneck filters are learned during backpropagation. The bottleneck also helps in reducing the depth after the pooling layer. The trick of bottleneck layers is also used in some other architectures, where it is helpful for improving efficiency and output depth.

The output layer of *GoogLeNet* also illustrates some interesting design principles. It is common to use fully connected layers near the output. However, *GoogLeNet* uses average pooling across the whole spatial area of the final set of activation maps to create a single value. Therefore, the number of features created in the final layer will be exactly equal to the number of filters. An important observation is that the vast majority of parameters are spent in connecting the final convolution layer to the first fully connected layer. This type of detailed connectivity is not required for applications in which only a class label needs to be predicted. Therefore, the average pooling approach is used. However, the average pooled representation completely loses all spatial information, and one must be careful of the types of applications it is used for. An important property of *GoogLeNet* was that it is extremely compact in terms of the number of parameters in comparison with *VGG*, and the number of parameters in the former is less by an order of magnitude. This is primarily because of the use of average pooling, which eventually became standard in many later architectures. On the other hand, the overall architecture of *GoogLeNet* is computationally more expensive.

The flexibility of *GoogLeNet* is inherent in the 22-layered inception architecture, in which objects of different scales are handled with the appropriate filter sizes. This flexibility of multigranular decomposition, which is enabled by the inception modules, was one of the keys to its performance. In addition, the replacement of the fully connected layer with average pooling greatly reduced the parameter footprint. This architecture was the winner of the ILSVRC contest in 2014, and *VGG* placed a close second. Even though *GoogLeNet* outperformed *VGG*, the latter does have the advantage of simplicity, which is sometimes appreciated by practitioners. Both architectures illustrated important design principles for convolution neural networks. The inception architecture has been the focus of significant research since then [502, 503], and numerous changes have been suggested to improve performance. In later years, a version of this architecture, referred to as *Inception-v4* [503], was combined with some of the ideas in *ResNet* (see next section) to create a 75-layer architecture with only 3.08% error.

[5]The original architecture also contained auxiliary classifiers, which have been ignored in recent years.

9.4.5 ResNet

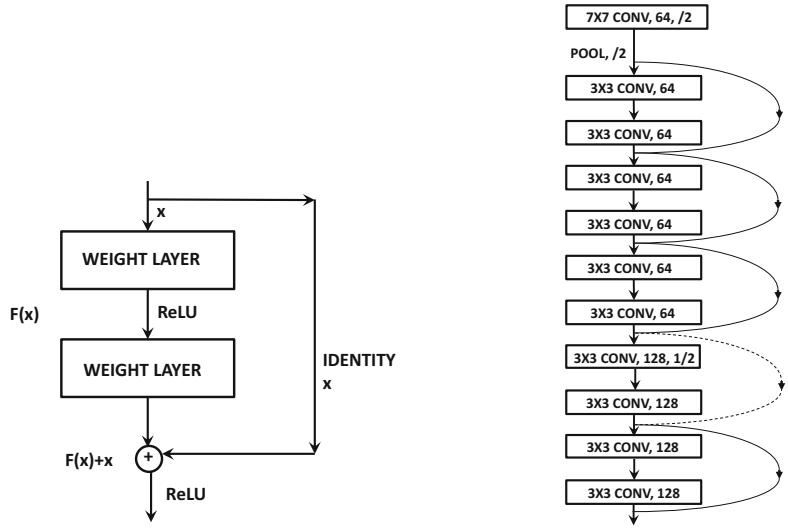
ResNet [194] used 152 layers, which was almost an order of magnitude greater than previously used by other architectures. This architecture was the winner of the ILSVRC competition in 2015, and it achieved a top-5 error of 3.6%, which resulted in the first classifier with human-level performance. This accuracy is achieved by an ensemble of *ResNet* networks; even a single model achieves 4.5% accuracy. Training an architecture with 152 layers is generally not possible unless some important innovations are incorporated.

The main issue in training such deep networks is that the gradient flow between layers is impeded by the large number of operations in deep layers that can increase or decrease the size of the gradients. As discussed in Chapter 4, problems such as the vanishing and exploding gradients are caused by increased depth. However, the work in [194] suggests that the main training problem in such deep networks might not necessarily be caused by these problems, especially if batch normalization is used. The main problem is caused by the difficulty in getting the learning process to converge properly in a reasonable amount of time. Such convergence problems are common in networks with complex loss surfaces. Although some deep networks show large gaps between training and test error, the error on both the training ad test data is high in many deep networks. This implies that the optimization process has not made sufficient progress.

Although hierarchical feature engineering is the holy grail of learning with neural networks, its layer-wise implementations force all concepts in the image to require the same level of abstraction. Some concepts can be learned by using shallow networks, whereas others require fine-grained connections. For example, consider a circus elephant standing on a square frame. Some of the intricate features of the elephant might require a large number of layers to engineer, whereas the features of the square frame might require very few layers. Convergence will be unnecessarily slow when one is using a very deep network with a fixed depth across all paths to learn concepts, many of which can also be learned using shallow architectures. Why not let the neural network decide how many layers to use to learn each feature?

ResNet uses *skip connections* between layers in order to enable copying between layers and introduces an *iterative view* of feature engineering (as opposed to a hierarchical view). Long short-term memory networks and gated recurrent units leverage similar principles in sequence data by allowing portions of the states to be copied from one layer to the next with the use of adjustable *gates*. In the case of *ResNet*, the non-existent "gates" are assumed to be always fully open. Most feed-forward networks only contain connections between layers i and $(i+1)$, whereas *ResNet* contains connections between layers i and $(i+r)$ for $r > 1$. Examples of such skip connections, which form the basic unit of *ResNet*, are shown in Figure 9.11(a) with $r = 2$. This skip connection simply copies the input of layer i and adds it to the output of layer $(i+r)$. Such an approach enables effective gradient flow because the backpropagation algorithm now has a super-highway for propagating the gradients backwards using the skip connections. This basic unit is referred to as a *residual module*, and the entire network is created by putting together many of these basic modules. In most layers, an appropriately padded filter[6] is used with a stride of 1, so that the spatial size and depth of the input does not change from layer to layer. In such cases, it is easy to simply add the input of the ith layer to that of $(i+r)$. However, some layers do use strided convolutions to reduce each spatial dimension by a factor of 2. At the same time, depth is increased by a factor of 2 by using a larger number of filters. In such a case, one cannot

[6]Typically, a 3×3 filter is used at a stride/padding of 1. This trend started with the principles in *VGG*, and was adopted by *ResNet*.



(a) Skip-connections in residual module (b) Partial architecture of *ResNet*

Figure 9.11: The residual module and the first few layers of *ResNet*

use the identity function over the skip connection. Therefore, a linear projection matrix might need to be applied over the skip connection in order to adjust the dimensionality. This projection matrix defines a set of 1×1 convolution operations with stride of 2 in order to reduce spatial extent by factor of 2. The parameters of the projection matrix need to be learned during backpropagation.

In the original idea of *ResNet*, one only adds connections between layers i and $(i+r)$. For example, if we use $r = 2$, only skip connections only between successive odd layers are used. Later enhancements like *DenseNet* showed improved performance by adding connections between all pairs of layers. The basic unit of Figure 9.11(a) is repeated in *ResNet*, and therefore one can traverse the skip connections repeatedly in order to propagate input to the output after performing very few forward computations. An example of the first few layers of the architecture is shown in Figure 9.11(b). This particular snapshot is based on the first few layers of the 34-layer architecture. Most of the skip connections are shown in solid lines in Figure 9.11(b), which corresponds to the use of the identity function with an unchanged filter volume. However, in some layers, a stride of 2 is used, which causes the spatial and depth footprint to change. In these layers, a projection matrix needs to be used, which is denoted by a dashed skip connection. Four different architectures were tested in the original work [194], which contained 34, 50, 101, and 152 layers, respectively. The 152-layer architecture had the best performance, but even the 34-layer architecture performed better than did the best-performing ILSVRC entry from the previous year.

The use of skip connections provides paths of unimpeded gradient flow and therefore has important consequences for the behavior of the backpropagation algorithm. The skip connections take on the function of super-highways in enabling gradient flow, creating a situation where multiple paths of variable lengths exist from the input to the output. In such cases, the shortest paths enable the most learning, and the longer paths can be viewed as residual contributions. This gives the learning algorithm the flexibility of choosing the appropriate level of nonlinearity for a particular input. Inputs that can be classified with a

small amount of nonlinearity will skip many connections. Other inputs with a more complex structure might traverse a larger number of connections in order to extract the relevant features. Therefore, the approach is also referred to as residual learning, in which learning along longer paths is a kind of fine tuning of the easier learning along shorter paths. In other words, the approach is well suited to cases in which different aspects of the image have different levels of complexity. The work in [194] shows that the residual responses from deeper layers are often relatively small, which validates the intuition that fixed depth is an impediment to proper learning. In such cases, the convergence is often not a problem, because the shorter paths enable a significant portion of the learning with unimpeded gradient flows. An interesting insight in [524] is that *ResNet* behaves like an ensemble of shallow networks because many alternative paths of shorter length are enabled by this type of architecture. Only a small amount of learning is enabled by the deeper paths, and only when it is absolutely necessary. The work in [524] in fact provides a pictorial depiction of an unraveled architecture of *ResNet* in which the different paths are explicitly shown in a parallel pipeline. This unraveled view provides a clear understanding of why *ResNet* has some similarities with ensemble-centric design principles. A consequence of this point of view is that dropping some of the layers from a trained *ResNet* at prediction time does not degrade accuracy as significantly as other networks like *VGG*.

More insights can be obtained by reading the work on *wide residual networks* [574]. This work suggests that increased depth of the residual network does not always help because most of the extremely deep paths are not used anyway. The skip connections do result in alternative paths and effectively increase the width of the network. The work in [574] suggests that better results can be obtained by limiting the total number of layers to some extent (say, 50 instead of 150), and using an increased number of filters in each layer. Note that a depth of 50 is still quite large from pre-*ResNet* standards, but is low compared to the depth used in recent experiments with residual networks. This approach also helps in parallelizing operations.

Variations of Skip Architectures

Since the architecture of *ResNet* was proposed, several variations were suggested to further improve performance. For example, the independently proposed *highway networks* [168] introduced the notion of gated skip connections, and can be considered a more general architecture. In highway networks, gates are used in lieu of the identity mapping, although a closed gate does not pass a lot of information through. In such cases, gating networks do not behave like residual networks. However, residual networks can be considered special cases of gating networks in which the gates are always fully open. Highway networks are closely related to both LSTMs and *ResNets*, although *ResNets* still seem to perform better in the image recognition task because of their focus on enabling gradient flow with multiple paths. The original *ResNet* architecture uses a simple block of layers between skip connections. However, the *ResNext* architecture varies on this principle by using inception modules between skip connections [559].

Instead of using skip connections, one can use convolution transformations between every pair of layers [222]. Therefore, instead of the L transformations in a feed-forward network with L layers, one is using $L(L-1)/2$ transformations. In other words, the concatenation of all the feature maps of the previous $(l-1)$ layers is used by the lth layer. This architecture is referred to as *DenseNet*. Note that the goal of such an architecture is similar to that of skip connections by allowing each layer to learn from whatever level of abstraction is useful.

An interesting variant that seems to work well is the use of *stochastic depth* [221] in which some of the blocks between skip connections are randomly dropped during training time, but the full network is used during testing time. Note that this approach seems similar to *Dropout*, which makes the network thinner rather than shallower by dropping nodes. However, *Dropout* has somewhat different motivations from layer-wise node dropping, because the latter is more focused on improving gradient flow rather than preventing feature co-adaptation.

9.4.6 Squeeze-and-Excitation Networks (SENets)

The winner of the image classification task in the ILSVRC competition of 2017 was the Squeeze-and-Excitation Network (SENet) [219]. The fundamental idea in SENet is to use a Squeeze-and-Excitation Block (SE Block), which could be *integrated with any existing network* between a pair of convolution operations. The SE Block first converts the qth spatial layer $H^q = [h^q_{ijp}]$ of size $L_q \times B_q \times d_q$ into a single vector of size d_q via average pooling on the *entire spatial region* in channel-wise fashion. This defines the *squeeze operation*. In other words, the vector $\overline{z} = [z_1, z_2, \ldots z_{d_q}]^T$ is defined as follows using the features h^q_{ijp} as follows:

$$z_p = \frac{\sum_{i=1}^{L_q} \sum_{j=1}^{B_q} h_{ijp}}{L_q \cdot B_q} \quad \forall p \in \{1 \ldots d_q\} \quad \text{[Average Pooling-Like Squeeze]}$$

Note that the vector $\overline{z}$ contains *channel-wise* average features. At this point the d_q-dimensional vector $\overline{z}$ is passed through a non-linear architecture with a single layer and reduction factor of r in order to allow the aggregate features of the different channels to interact with one another with the use of learned weights. This is the *excitation* operation and its uses a learned matrix W of size $d_q \times d_q/r$ in order to create a new vector $\overline{z}'$ of size d_q:

$$\overline{z}' = \text{Sigmoid}\left(W[\text{ReLU}(W^T \overline{z})]\right) \quad \text{[Learning Channel Weights]}$$

The constricted architecture learns the weights of channels based on distribution of intensity across different channels. The approach is particularly important for later layers and is typically not applied to the input layer (even though Figure 9.12 provides a notional example of color contrasts). *The features emphasized will be sensitive to the task (e.g., image classification), data set, and input image characteristics (average channel intensities).* Similarly, nose features are more important than window features in facial recognition. This layer is referred to as an "excitation" layer, as it provides weights for different channels depending on their informativeness. Therefore, these d_q weights of $\overline{z}' = [z'_1, z'_2, \ldots z'_{d_q}]^T$ will be used in order to scale each of the d_q channels in the original spatial feature representation H_q to create the enhanced features $H'_q = [h'_{ijp}]$ after re-weighting:

$$h'_{ijp} = h_{ijp} \cdot z'_p \quad \forall i, j, p \quad \text{[Channel-wise Scaling of Informative Features]}$$

The overall architecture of the SE Block is shown in Figure 9.12. The value of the constriction factor r is typically chosen to be around 16.

The SE Blocks perform feature engineering across all layers by using the aggregate statistics of the features to properly scale the channel-wise representations, so that more informative features tend to be emphasized by the scaling. The idea of emphasizing some features in this way is also referred to as an *attention mechanism* (see section 12.2 of Chapter 12). The role of this type of scaling differs throughout the network (as the SE

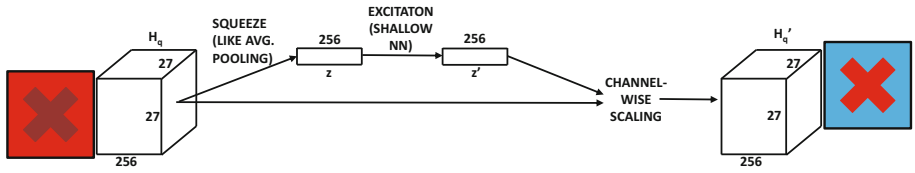


Figure 9.12: The SE Block

Table 9.3: The number of layers in various top-performing ILSVRC contest entries

Name	Year	Number of Layers	Top-5 Error
-	Before 2012	≤ 5	$> 25\%$
AlexNet	2012	8	15.4%
ZfNet/Clarifai	2013	8/> 8	14.8% / 11.1%
VGG	2014	19	7.3%
GoogLeNet	2014	22	6.7%
ResNet	2015	152	3.6%

Block may be used across all layers). In earlier layers, informative features are emphasized in a class-independent manner, whereas in later layers class-specific features tend to be emphasized. The beauty of the SENet architecture is that it can be integrated with any of the earlier architectures, such as VGG, inception modules, or ResNet. In that sense, the SE Blocks provide an additional performance improvement over a base model much like a meta-learner or ensemble. The best results for SENets were obtained in the ILSVRC competition by using a variant of ResNet in which a top-5 error rate of only 2.25% was obtained.

9.4.7 The Effects of Depth

The significant advancements in performance in recent years in the ILSVRC contest are mostly a result of improved computational power, greater data availability, and changes in architectural design that have enabled the effective training of neural networks with increased depth. These three aspects also support each other, because experimentation with better architectures is only possible with sufficient data and improved computational efficiency. This is also one of the reasons why the fine-tuning and tweaks of relatively old architectures (like recurrent neural networks) with known problems were not performed until recently.

The number of layers and the error rates of various networks are shown in Table 9.3. The rapid increase in accuracy in the short period from 2012 to 2015 is quite remarkable, and is unusual for most machine learning applications that are as well studied as image recognition. Another important observation is that increased depth of the neural network is closely correlated with improved error rates. Therefore, an important focus of the research in recent years has been to enable algorithmic modifications that support increased depth. It is noteworthy that convolutional neural networks are among the deepest of all classes of neural networks. Interestingly, traditional feed-forward networks in other domains do not need to be very deep for most applications like classification. Indeed, the coining of the term "deep learning" owes a lot of its origins to the impressive performances of convolutional neural networks and specific improvements observed with increased depth. The improvements in 2016 and 2017 were not obtained with increased depth but with tricks such as ensembles and the use of Squeeze-and-Excitation Blocks.

9.4.8 Pretrained Models

One of the challenges faced by analysts in the image domain is that *labeled* training data may not even be available for a particular application. Consider the case in which one has a set of images that need to be used for image retrieval. In retrieval applications, labels are not available but it is important for the features to be semantically coherent. In some other cases, one might wish to perform classification on a smaller data set with a particular set of labels, which might be different from that of *ImageNet*. These settings cause problems because neural networks require a lot of training data to build from scratch.

However, a key point about image data is that the extracted features from a particular data set are highly reusable across data sources. For example, the way in which a cat is represented will not vary a lot if the same number of pixels and color channels are used in different data sources. In such cases, generic data sources, which are representative of a wide spectrum of images, are useful. For example, the *ImageNet* data set [605] contains more than a million images drawn from 1000 categories encountered in everyday life. The chosen 1000 categories and the large diversity of images in the data set are representative and exhaustive enough that one can use them to extract features of images for general-purpose settings. For example, the features extracted from the *ImageNet* data can be used to represent a completely different image data set by passing it through a pretrained convolutional neural network (like *AlexNet*) and extracting the multidimensional features from the fully connected layers. This new representation can be used for a completely different application like clustering or retrieval. This type of approach is so common, that *one rarely trains convolutional neural networks from scratch*. The extracted features from the penultimate layer are often referred to as FC7 features, which is an inheritance from the name of the layer in *AlexNet*. Of course, an arbitrary convolutional network might not have the same number of layers as *AlexNet*; however, the name FC7 has stuck.

This type of off-the-shelf feature extraction approach [406] can be viewed as a kind of *transfer learning*, because we are using a public resource like *ImageNet* to extract features to solve different problems in settings where enough training data is not available. Such an approach has become standard practice in many image recognition tasks, and many software frameworks like *Caffe* provide ready access to these features [609, 610]. In fact, *Caffe* provides a "zoo" of such pretrained models, which can be downloaded and used [610]. If some additional training data is available, one can use it to fine-tune only the deeper layers (i.e., layers closer to the output layer). The weights of the early layers (closer to the input) are fixed. The reason for training only the deeper layers, while keeping the early layers fixed, is that the earlier layers capture only primitive features like edges, whereas the deeper layers capture more complex features. The primitive features do not change too much with the application at hand, whereas the deeper features might be sensitive to the application at hand. For example, all types of images will require edges of different orientation to represent them (captured in early layers), but a feature corresponding to the wheel of a truck will be relevant to a data set containing images of trucks. In other words, early layers tend to capture highly generalizable features (across different computer vision data sets), whereas later layers tend to capture data-specific features. A discussion of the transferability of features derived from convolutional neural networks across data sets and tasks is provided in [372].

9.5 Visualization and Unsupervised Learning

An interesting property of convolutional neural networks is that they are highly interpretable in terms of the types of features they can learn. However, it takes some effort to actually interpret these features. The first approach that comes to mind is to simply visualize the 2-dimensional (spatial) components of the filters. Although this type of visualization can provide some interesting visualizations of the primitive edges and lines learned in the first layer of the neural network, it is not very useful for later layers. In the first layer, it is possible to visualize these filters because they operate directly on the input image, and often tend to look like primitive parts of the image (such as edges). However, it is not quite as simple a matter to visualize these filters in later layers because they operate on input volumes that have already been scrambled with convolution operations. In order to obtain any kind of interpretability one must find a way to map the impacts of all operations all the way back to the input layer. Therefore, the goal of visualization is often to identify and highlight the portions of the input image to which a particular hidden feature is responding. For example, the value of one hidden feature might be sensitive to changes in the portion of the image corresponding to the wheel of a truck, and a different hidden feature might be sensitive to its hood. This is naturally achieved by computing the sensitivity (i.e., gradient) of a hidden feature with respect to each pixel of the input image. As we will see, these types of visualizations are closely related to backpropagation, unsupervised learning, and transposed convolutional operations (used for creating the decoder portions of autoencoders). Therefore, this chapter will discuss these closely related topics in an integrated way.

There are two primary settings in which one can encode and decode an image. In the first setting, the compressed feature maps are learned by using any of the supervised models discussed in earlier sections. Once the network has been trained in a supervised way, one can attempt to reconstruct the portions of the image that most activate a given feature. Furthermore, the portions of an image that are most likely to activate a particular hidden feature or a class are identified. As we will see later, this goal can be achieved with various types of backpropagation and optimization formulations. The second setting is purely unsupervised, in which a convolutional network (encoder) is hooked up to a deconvolutional network (decoder). As we will see later, the latter is also a form of transposed convolution, which is similar to backpropagation. However, in this case, the weights of the encoder and decoder are learned jointly to minimize the reconstruction error. The first setting is obviously simpler because the encoder is trained in a supervised way, and one only has to learn the effect of different portions of the input field on various hidden features. In the second setting, the entire training and learning of weights of the network has to be done from scratch.

9.5.1 Visualizing the Features of a Trained Network

Consider a neural network that has already been trained using a large data set like *ImageNet*. The goal is to visualize and understand the impact of the different portions of the input image (i.e., receptive field) on various features in the hidden layers and the output layer (e.g., the 1000 softmax outputs in *AlexNet*). We would like to answer the following questions:

1. Given an activation of a feature anywhere in the neural network for a *particular* input image, visualize the portions of the input to which that feature is responding the most. Note that the feature might be one of the hidden features in the spatially arranged layers, in the fully connected hidden layers (e.g., FC7), or even one of the softmax outputs. In the last of these cases, one obtains some insight of the specific relationship

of a particular input image to a class. For example, if an input image is activating the label for "*banana*," we hope to see the parts of the specific input image that look most like a banana.

2. Given a particular feature anywhere in the neural network, find a fantasy image that is likely to activate that feature the most. As in the previous case, the feature might be one of the hidden features or even one of the features from the softmax outputs. For example, one might want to know what type of fantasy image is most likely to classify to a "*banana*" in the trained network at hand.

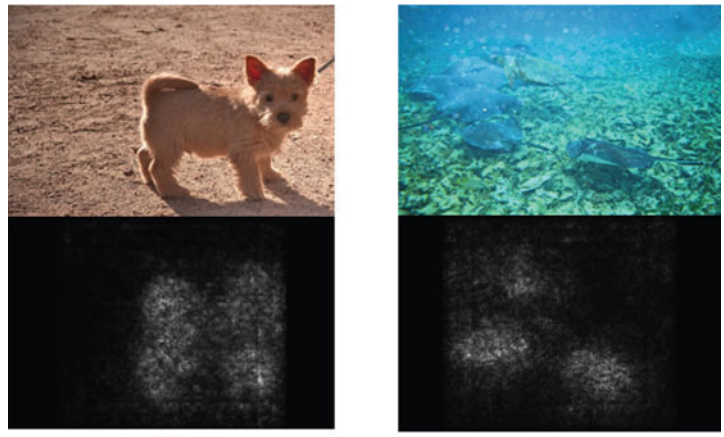
In both these cases, the easiest approach to visualize the impact of specific features is to use gradient-based methods. The second of the above goals is rather hard, and one often does not obtain satisfactory visualizations without careful regularization.

Gradient-Based Visualization of Activated Features

The backpropagation algorithm that is used to train the neural network is also helpful for gradient-based visualization. It is noteworthy that backpropagation-based gradient computation is a form of transposed convolution. In traditional autoencoders, transposed weight matrices (of those used in the encoder layer) are often used in the decoder. Therefore, the connections between backpropagation and feature reconstruction are deep and are applicable across all types of neural networks. The main difference from the traditional backpropagation setting is that our end-goal is to determine the sensitivity of the hidden/output features with respect to *different pixels of the input image* rather than with respect to the weights. However, even traditional backpropagation does compute the sensitivity of the outputs with respect to various layers as an intermediate step, and therefore almost exactly the same approach can be used in both cases.

When the sensitivity of an output o is computed with respect to the input pixels, the visualization of this sensitivity over the corresponding pixels is referred to as a *saliency map* [476]. For example, the output o might be the softmax probability (or unnormalized score before applying softmax) of the class "*banana*." Then, for each pixel x_i in the image, we would like to determine the value of $\frac{\partial o}{\partial x_i}$. This value can be computed by straightforward backpropagation all the way[7] to the input layer. The softmax probability of "*banana*" will be relatively insensitive to small changes in those portions of the image that are irrelevant to the recognition of a banana. Therefore, the values of $\frac{\partial o}{\partial x_i}$ will be close to 0 for such irrelevant regions, whereas the portions of the image that define a banana will have large magnitudes. For example, in the case of *AlexNet*, the entire $224 \times 224 \times 3$ volume defined by $\frac{\partial o}{\partial x_i}$ of *backpropagated gradients* will have portions with large magnitudes corresponding to the banana in the image. To visualize this volume, we first convert it to grayscale by taking the *maximum of the absolute magnitude of the gradient* over the three RGB channels to create a $224 \times 224 \times 1$ map with only non-negative values. The bright portions of this grayscale visualization will tell us which portion of the input image are relevant to the banana. Examples of grayscale visualizations of the portions of the image that excite relevant classes are shown in Figure 9.13. For example, the bright portion of the image in Figure 9.13(a) excites the animal in the image, which also represents its class label.

[7]Under normal circumstances, one only backpropagates to hidden layers as an intermediate step to compute gradients with respect to incoming weights in that hidden layer. Therefore, backpropagation to input layer is never really needed in traditional training. However, backpropagation to the input layer is identical to that with respect to the hidden layers.



(a) (b)

Figure 9.13: Examples of portions of specific images activated by particular class labels. These images appear in the work by Simonyan, Vedaldi, and Zisserman [476]. Reproduced with permission. (©2014 Simonyan, Vedaldi, and Zisserman)

This general approach has also been used for visualizing the activations of specific hidden features. Consider the value h of a hidden variable for a particular input image. How is this variable responding to the input image at its current activation level? The idea is that if we slightly increase or decrease the color intensity of some pixels, the value of h will be affected more than if we increase or decrease other pixels. First, the hidden variable h will be affected by a small rectangular portion of the image (i.e., receptive field), which is very small when h is present in early layers but much larger in later layers. For example, the receptive field of h might only be of size 3×3 when it is selected from the first hidden layer in the case of *VGG*. Examples of the image crops corresponding to specific images in which a particular neuron in a hidden layer is highly activated are shown in each row on the right-hand side of Figure 9.14. Note that each row contains a somewhat similar image. This is not a coincidence because that row corresponds to a particular hidden feature, and the variations in that row are caused by the different choices of image. Note that the choices of the image for a row is also not random, because we are selecting the images that most activate that feature. Therefore, all the images will contain the same visual characteristic that cause this hidden feature to be activated. The grayscale portion of the visualization corresponds to the sensitivity of the feature to the pixel-specific values in the corresponding image crop.

At a high level of activation level of h, some of the pixels in that receptive field will be more sensitive to h than others. By isolating the pixels to which the hidden variable h has the greatest sensitivity and visualizing the corresponding rectangular regions, one can get an idea of what part of the input map most affects a particular hidden feature. Therefore, any particular pixel x_i, we want to compute $\frac{\partial h}{\partial x_i}$, and then visualize those pixels with large values of this gradient. However, instead of backpropagation, the notions of *"deconvnet"* [581] and *guided backpropagation* [482] are sometimes used. The notion of "deconvent" is also used in convolutional autoencoders. The main difference is in terms of how the gradient of the ReLU nonlinearity is propagated backwards. As discussed in Table 2.1 of Chapter 2, the partial derivative of a ReLU unit is copied backwards during backpropagation if the *input* to the

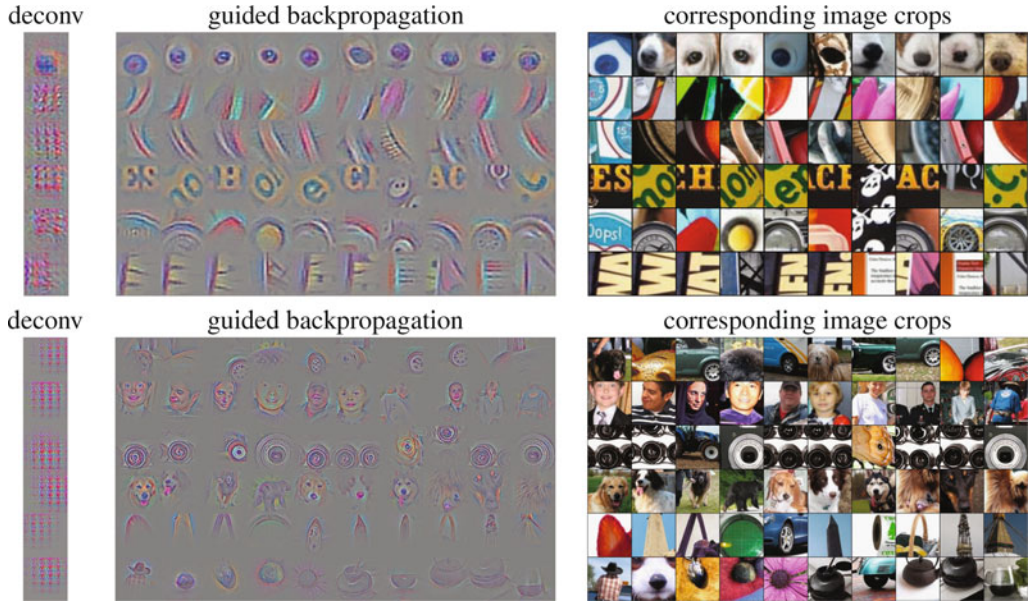


Figure 9.14: Examples of activation visualizations in different layers in Springenberg *et al.*'s work [482]. Reprinted from [482] with permission (©2015 Springenberg, Dosovitskiy, Brox, Riedmiller).

ReLU is positive, and is otherwise set to 0. However, in "deconvnet," the partial derivative of a ReLU unit is copied backwards, if this partial derivative is itself larger than 0. This is like using a ReLU on the propagated gradient in the backward pass. In other words, we replace $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$ in Table 2.1 with $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{g}_{i+1} > 0)$. Here $\overline{z}_i$ represents the forward activations, and $\overline{g}_i$ represents the backpropagated gradients with respect to the ith layer containing only ReLU units. The function $I(\cdot)$ is an element-wise indicator function, which takes on the value of 1 for each element in the vector argument when the condition is true for that element. In guided backpropagation, we *combine* the conditions used in traditional backpropagation and ReLU by using $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0) \odot I(\overline{g}_{i+1} > 0)$. A pictorial illustration of the three variations of backpropagation is shown in Figure 9.15. It is suggested in [482] that guided backpropagation gives better visualizations than "deconvnet," which in turn gives better results than traditional backpropagation.

One way of interpreting the difference between traditional backpropagation and "deconvnet" is by interpreting backwards propagation of gradients as the operations of a decoder with transposed convolutions with respect to the encoder [476]. However, in this decoder, we are again using the ReLU function rather than the gradient-based transformation implied by the ReLU. After all, all forms of decoders use the same activation functions as the encoder. Another feature of the visualization approach in [482] is that it omits the use of pooling layers in the convolutional neural network altogether, and instead relies on strided convolutions. The work in [482] identified several highly activated neurons in specific layers corresponding to specific input images and provided visualizations of the rectangular regions of those images corresponding to the receptive fields of those hidden neurons. We have already discussed earlier that the right-hand side of Figure 9.14 contains the input regions corresponding to specific neurons in hidden layers. The left-hand side of Figure 9.14

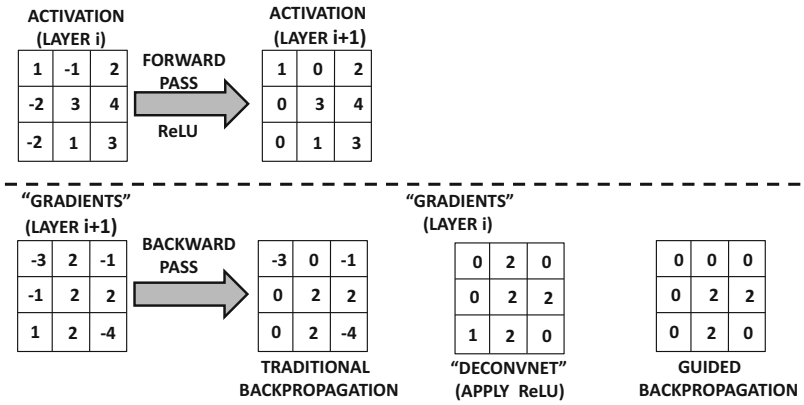


Figure 9.15: The different variations of backpropagation of ReLU for visualization

also shows the specific characteristics of each image that excite that particular neuron. The visualization on the left-hand side is obtained with guided backpropagation. Note that the upper set of images correspond to the sixth layer, whereas the lower set of images corresponds to the ninth layer of the convolutional network. As a result, the images in the lower set correspond to larger regions of the input image containing more complex shapes.

Another excellent set of visualizations from [581] is shown in Figure 9.16. The main difference is that the work in [581] also uses max-pooling layers, and is based on deconvolutions rather than guided backpropagation. The specific hidden variables chosen are the top-9 largest activations in each feature map. In each case, the relevant square region of the image is shown together with the corresponding visualization. It is evident that the hidden features in early layers correspond to primitive lines, which become increasingly more complex in later layers. This is one of the reasons that convolutional neural networks are viewed as methods that create hierarchical features. The features in early layers tend to be more generic, and they can be used across a wider variety of data sets. The features in later layers tend to be more specific to individual data sets. This is a key property exploited in transfer learning applications, in which pretrained networks are broadly used, and only the later layers are fine-tuned in a manner that's specific to data set and application.

Synthesized Images that Activate a Feature

The above examples tell us the portions of a *particular* image that most affect a particular neuron. A more general question is to ask what kind of image patch would maximally activate a particular neuron. For ease in discussion, we will discuss the case in which the neuron is an output value o of a particular class (i.e., unnormalized output before applying softmax). For example, the value of o might be the unnormalized score for "*banana*." Note that one can also apply a similar approach to intermediate neurons rather than the class score. We would like to learn the input image $\overline{x}$ that maximizes the output o, while applying regularization to $\overline{x}$:

$$\text{Maximize}_{\overline{x}} \; J(\overline{x}) = (o - \lambda ||\overline{x}||^2)$$

Here, λ is the regularization parameter, and is important in order to extract semantically interpretable images. One can use gradient ascent in conjunction with backpropagation in order to learn the input image $\overline{x}$ that maximizes the above objective function. Therefore,

Figure 9.16: Examples of activation visualizations in different layers based on Zeiler and Fergus's work [581]. Reprinted from [581] with permission. ©Springer International Publishing Switzerland, 2014.

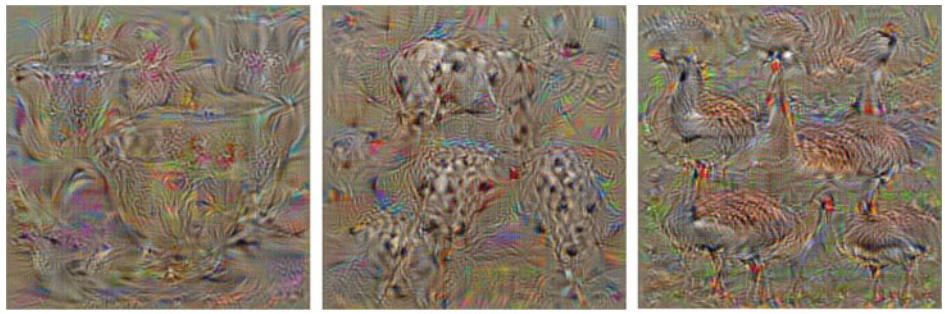


Figure 9.17: Examples of synthesized images with respect to particular class labels. These examples appear in the work by Simonyan, Vedaldi, and Zisserman [476]. Reproduced with permission (©2014 Simonyan, Vedaldi, and Zisserman)

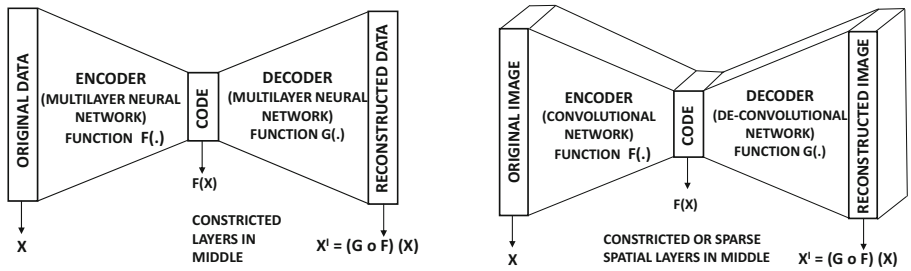
we start with a zero image $\overline{x}$ and update $\overline{x}$ using gradient ascent in conjunction with backpropagation with respect to the above objective function. In other words, the following update is used:

$$\overline{x} \Leftarrow \overline{x} + \alpha \nabla_{\overline{x}} J(\overline{x}) \tag{9.2}$$

Here, α is the learning rate. The key point is that backpropagation is being leveraged in an unusual way to update the *image pixels* while keeping the (already learned) weights fixed. Examples of synthesized images for three classes are shown in Figure 9.17. Other advanced methods for generating more realistic images on the basis of class labels are discussed in [369].

9.5.2 Convolutional Autoencoders

The use of the autoencoder in traditional neural networks is discussed in Chapters 3 and 5. Recall that the autoencoder reconstructs data points after passing them through a compression phase. In some cases, the data is not compressed although the representations are sparse. The portion before the most compressed layer of the architecture is referred to as the encoder, and the portion after the compressed portion is referred to as the decoder. We repeat the pictorial view of the encoder-decoder architecture for the traditional case in Figure 9.18(a). The convolutional autoencoder has a similar principle, which reconstructs images after passing them through a compression phase. The main difference between a traditional autoencoder and a convolutional autoencoder is that the latter is focused on using spatial relationships between points in order to extract features that have a visual interpretation. The spatial convolution operations in the intermediate layers achieve precisely this goal. An illustration of the convolutional autoencoder is shown in Figure 9.18(b) in comparison with the traditional autoencoder in Figure 9.18(a). Note the 3-dimensional spatial shape of the encoder and decoder in the second case. However, it is possible to conceive of several variations to this basic architecture. For example, the codes in the middle can either be spatial or they can be flattened with the use of fully connected layers, depending on the application at hand. The fully connected layers would be necessary to create a multidimensional code that can be used with arbitrary applications (without worrying about spatial constraints among features). In the following, we will simplify the discussion by assuming that the compressed code in the middle is spatial in nature.



(a) A traditional autoencoder architecture (b) Convolutional autoencoder

Figure 9.18: A traditional autoencoder and a convolutional autoencoder.

Table 9.4: The relationship between backpropagation and decoders

Linear Operation	Traditional neural networks	Convolutional neural networks
Forward Propagation	Matrix multiplication	Convolution
Backpropagation	Transposed matrix multiplication	Transposed convolution
Decoder layer	Transposed matrix multiplication (Identical to backpropagation)	Transposed convolution (Identical to backpropagation)

Just as the compression portion of the encoder uses a convolution operation, the decompression operation uses a *deconvolution* operation. Similarly, pooling is matched with an *unpooling* operation. Deconvolution is also referred to as *transposed convolution*. Interestingly, the transposed convolution operation is the same as that used for backpropagation. The term "deconvolution" is perhaps a bit misleading because every deconvolution is in actuality a convolution with a filter that is derived by transposing and inverting the tensor representing the original convolution filter (cf. Figure 9.7 and Equation 9.1). We can already see that deconvolution uses similar principles to that of backpropagation. The main difference is in terms of how the ReLU function is handled, which makes deconvolution more similar to "deconvnet" or *guided* backpropagation. In fact, the decoder in a convolutional autoencoder performs similar operations to the backpropagation phase of gradient-based visualization. Some architectures do away with the pooling and unpooling operations, and work with only convolution operations (together with activation functions) [469, 482].

The fact that the deconvolution operation is really not much different from a convolution operation is not surprising. Even in traditional feed-forward networks, the decoder part of the network performs the same types of matrix multiplications as the encoder part of the network, except that the transposed weight matrices are used. One can summarize the analogy between traditional autoencoders and convolutional autoencoders in Table 9.4. Note that the relationship between forward backpropagation and backward propagation is similar in traditional and convolutional neural networks in terms of how the corresponding matrix operations are performed. A similar observation is true about the nature of the relationship between encoders and decoders.

There are three operations corresponding to the convolution, max-pooling, and the ReLU nonlinearity. The goal is to perform the inversion of the operations in the decoder layer that have been performed in the encoder layer. There is no easy way to exactly invert some of the

operations (such as max-pooling and ReLU). However, excellent image reconstruction can still be achieved with the proper design choices. First, we describe the case of an autoencoder with a single layer with convolution, ReLU, and max-pooling. Then, we discuss how to generalize it to multiple layers.

Although one typically wants to use the inverse of the encoder operations in the decoder, the ReLU is not an invertible function because a value of 0 has many possible inversions. Therefore, a ReLU is replaced by another ReLU in the decoder layer (although other options are possible). Therefore, the architecture of this simple autoencoder is as follows:

$$\underbrace{\text{Convolve} \Rightarrow \text{ReLU} \Rightarrow \text{Max-Pool}}_{\text{Encoder}} \Rightarrow \text{Code} \Rightarrow \underbrace{\text{Unpool} \Rightarrow \text{ReLU} \Rightarrow \text{De-Convolve}}_{\text{Decoder}}$$

Note that the layers are symmetrically arranged in terms of how a matching layer in the decoder undoes the effect of a corresponding layer in the encoder. However, there are many variations to this basic theme. For example, the ReLU might be placed after the deconvolution. Furthermore, in some variations [322], it is recommended to use deeper encoders than the decoders with an asymmetric architecture. However, with a stacked variation of the symmetric architecture above, it is possible to train just the encoder with a classification output layer (and a supervised data set like *ImageNet*) and then use its symmetric decoder (with transposed/inverted filters) to perform "deconvnet" visualization [581]. Although one can always use this approach to initialize the autoencoder, we will discuss enhancements of this concept where the encoder and decoder are jointly trained in an unsupervised way.

We will count each layer like convolution and ReLU as a separate layer here, and therefore we have a total of seven layers including the input. This architecture is simplistic because it uses a single convolution layer in each of the encoders and decoders. In more generalized architectures, these layers are stacked to create more powerful architectures. However, it is helpful to illustrate the relationship of the basic operations like unpooling and deconvolution to their encoding counterparts (like pooling and convolution). Another simplification is that the code is contained in a spatial layer, whereas one could also insert fully connected layers in the middle. Although this example (and Figure 9.18(b)) uses a spatial code, the use of fully connected layers in the middle is more useful for practical applications. On the other hand, the spatial layers in the middle can be used for visualization.

Consider a situation in which the encoder uses d_2 square filters of size $F_1 \times F_1 \times d_1$ in the first layer. Also assume that the first layer is a (spatially) square volume of size $L_1 \times L_1 \times d_1$. The (i, j, k)th entry of the pth filter in the first layer has weight $w_{ijk}^{(p,1)}$. This notations are consistent with those used in section 9.2, where the convolution operation is defined. It is common to use the precise level of padding required in the convolution layer, so that the feature maps in the second layer are also of size L_1. This level of padding is $F_1 - 1$, which is referred to as *half-padding*. However, it is also possible to use no padding in the convolution layer, if one uses full padding in the corresponding deconvolution layer. In general, the sum of the paddings between the convolution and its corresponding deconvolution layer must sum to $F_1 - 1$ in order to maintain the spatial size of the layer in a convolution-deconvolution pair.

Here, it is important to understand that although each $W^{(p,1)} = [w_{ijk}^{(p,1)}]$ is a 3-dimensional tensor, one can create a 4-dimensional tensor by including the index p in the tensor. The deconvolution operation uses a transposition of this tensor, which is similar to the approach used in backpropagation (cf. section 9.3.3). The counter-part deconvolution operation occurs from the sixth to the seventh layer (by counting the ReLU/pooling/unpooling layers in the middle). Therefore, we will define the (deconvolution) tensor $U^{(s,6)} = [u_{ijk}^{(s,6)}]$ in

relation to $W^{(p,1)}$. Layer 5 contains d_2 feature maps, which were inherited from the convolution operation in the first layer (and unchanged by pooling/unpooling/ReLU operations). These d_2 feature maps need to be mapped into d_1 layers, where the value of d_1 is 3 for RGB color channels. Therefore, the number of filters in the deconvolution layer is equal to the depth of the filters in the convolution layer and vice vera. One can view this change in shape as a result of the transposition and spatial inversion of the 4-dimensional tensor created by the filters. Furthermore, the entries of the two 4-dimensional tensors are related as follows:

$$u_{ijk}^{(s,6)} = w_{rms}^{(k,1)} \quad \forall s \in \{1 \ldots d_1\}, \forall k \in \{1 \ldots d_2\} \tag{9.3}$$

Here, $r = n - i + 1$ and $m = n - j + 1$, where the spatial footprint in the first layer is $n \times n$. Note the transposition of the indices s and k in the above relationship. This relationship is identical to Equation 9.1. It is not necessary to tie the weights in the encoder and decoder, or even use a symmetric architecture between encoder and decoder [322].

The filters $U^{(s,6)}$ in the sixth layer are used just like any other convolution to reconstruct the RGB color channels of the images from the activations in layer 6. Therefore, a deconvolution operation is really a convolution operation, except that it is done with a transposed and spatially inverted filter. As discussed in section 9.3.2, this type of deconvolution operation is also used in backpropagation. Both the convolution/deconvolution operations can also be executed with the use of matrix multiplications, as described in that section.

The pooling operations, however, irreversibly lose some information and are therefore impossible to invert exactly. This is because the non-maximal values in the layer are permanently lost by pooling. The max-unpooling operation is implemented with the help of *switches*. When pooling is performed, the precise positions of the maximal values are stored. For example, consider the common situation in which 2×2 pooling is performed at stride 2. In such a case, pooling reduces both spatial dimensions by a factor of 2, and it picks the maximum out of $2 \times 2 = 4$ values in each (non-overlapping) pooled region. The exact coordinate of the (maximal) value is stored, and is referred to as the switch. When unpooling, the dimensions are increased by a factor of 2, and the values at the switch positions are copied from the previous layer. The other values are set to 0. Therefore, after max-unpooling, exactly 75% of the entries in the layer will have uncopied values of 0 in the case of non-overlapping 2×2 pooling.

Like traditional autoencoders, the loss function is defined by the reconstruction error over all $L_1 \times L_1 \times d_1$ pixels. Therefore, if $h_{ijk}^{(1)}$ represents the values of the pixels in the first (input) layer, and $h_{ijk}^{(7)}$ represents the values of the pixels in the seventh (output) layer, the reconstruction loss E is defined as follows:

$$E = \sum_{i=1}^{L_1} \sum_{j=1}^{L_1} \sum_{k=1}^{d_1} (h_{ijk}^{(1)} - h_{ijk}^{(7)})^2 \tag{9.4}$$

Other types of error functions (such as L_1-loss and negative log-likelihood) are also used.

One can use traditional backpropagation with the autoencoder. Backpropagating through deconvolutions or the ReLU is no different than in the case of convolutions. In the case of max-unpooling, the gradient flows only through the switches in an unchanged way. Since the parameters of the encoder and the decoder are tied, one needs to sum up the gradients of the matching parameters in the two layers during gradient descent. Another interesting point is that backpropagating through deconvolutions uses almost identical operations to forward propagation through convolutions. This is because both backpropagation and deconvolution cause successive transpositions of the 4-dimensional tensor used for transformation.

This basic autoencoder can easily be extended to the case where multiple convolutions, poolings, and ReLUs are used. The work in [579] discusses the difficulty with multilayer autoencoders, and proposes several tricks to improve performance. There are several other architectural design choices that are often used to improve performance. One key point is that strided convolutions are often used (in lieu of max-pooling) to reduce the spatial footprint in the encoder, which must be balanced in the decoder with *fractionally* strided convolutions. Consider a situation in which the encoder uses a stride of S with some padding to reduce the size of the spatial footprint. In the decoder, one can increase the size of the spatial footprint by the same factor by using the following trick. While performing the convolution, we stretch the input volume by placing $S - 1$ rows of zeros[8] between every pair of rows, and $S - 1$ columns of zeros between every pair of columns before applying the filter. As a result, the input volume already stretches by a factor of approximately S in each spatial dimension. Additional padding along the borders can be applied before performing the convolution with the transposed filter. Such an approach has the effect of providing a fractional stride and expanding the output size in the decoder. An alternative approach for stretching the input volume of a convolution is to insert interpolated values (instead of zeros) between the original entries of the input volume. The interpolation is done using a convex combination of the nearest four values, and a decreasing function of the distance to each of these values is used as the proportionality factor of the interpolation [469]. The approach of stretching the inputs is sometimes combined with that of stretching the filters as well by inserting zeros within the filter [469]. Stretching the filter results in an approach, referred to as *dilated convolution*, although its use is not universal for fractionally strided convolutions. A detailed discussion of convolution arithmetic (included fractionally strided convolution) is provided in [111]. Compared to the traditional autoencoder, the convolutional autoencoder is somewhat more tricky to implement, with many different variations for better performance. Refer to the bibliographic nodes.

Unsupervised methods also have applications to improving supervised learning. The most obvious method among them is *pretraining*, which was discussed in section 5.7 of Chapter 5. In convolutional neural networks, the methodology for pretraining is not very different in principle from what is used in traditional neural networks. Pretraining can also be performed by deriving the weights from a trained *deep-belief convolutional network* [291]. This is analogous to the approach in traditional neural networks, where stacked Boltzmann machines were among the earliest models used for pretraining.

9.6 Applications of Convolutional Networks

Convolutional neural networks have several applications in object detection, localization, video, and text processing. Many of these applications work on the basic principle of using convolutional neural networks to provide engineered features, on top of which multidimensional applications can be constructed. The success of convolutional neural networks remains unmatched by almost any class of neural networks. In recent years, competitive methods have even been proposed for sequence-to-sequence learning, which has traditionally been the domain of recurrent networks.

[8]Example available at http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html.

9.6.1 Content-Based Image Retrieval

In content-based image retrieval, each image is first engineered into a set of multidimensional features by using a pretrained classifier like *AlexNet*. The pretraining is typically done up front using a large data set like *ImageNet*. A huge number of choices of such pretrained classifiers is available at [610]. The features from the fully connected layers of the classifier can be used to create a multidimensional representation of the images. The multidimensional representations of the images can be used in conjunction with any multidimensional retrieval system to provide results of high quality. The use of neural codes for image retrieval is discussed in [17]. The reason that this approach works is because the features extracted from *AlexNet* have semantic significance to the different types of shapes present in the data. As a result, the quality of the retrieval is generally quite high when working with these features.

9.6.2 Object Localization

In object localization, we have a fixed set of objects in an image, and we would like to identify the rectangular regions in the image in which the object occurs. The basic idea is to take an image with a *fixed* number of objects and encase each of them in a bounding box. In the following, we will consider the simple case in which a single object exists in the image. Image localization is usually integrated with the classification problem, in which we first wish to classify the object in the image and draw a bounding box around it. For simplicity, we consider the case in which there is a single object in the image. We have shown an example of image classification and localization in Figure 9.19, in which the class "*fish*" is identified, and a bounding box is drawn around the portion of the image that delineates that class.

The bounding box of an image can be uniquely identified with four numbers. A common choice is to identify the top-left corner of the bounding box, and the two dimensions of the box. Therefore, one can identify a box with four unique numbers. This is a regression problem with multiple targets. Here, the key is to understand that one can train almost the same model for both classification and regression, which vary only in terms of the final

Figure 9.19: Example of image classification/localization in which the class "*fish*" is identified together with its bounding box. The image is illustrative only.

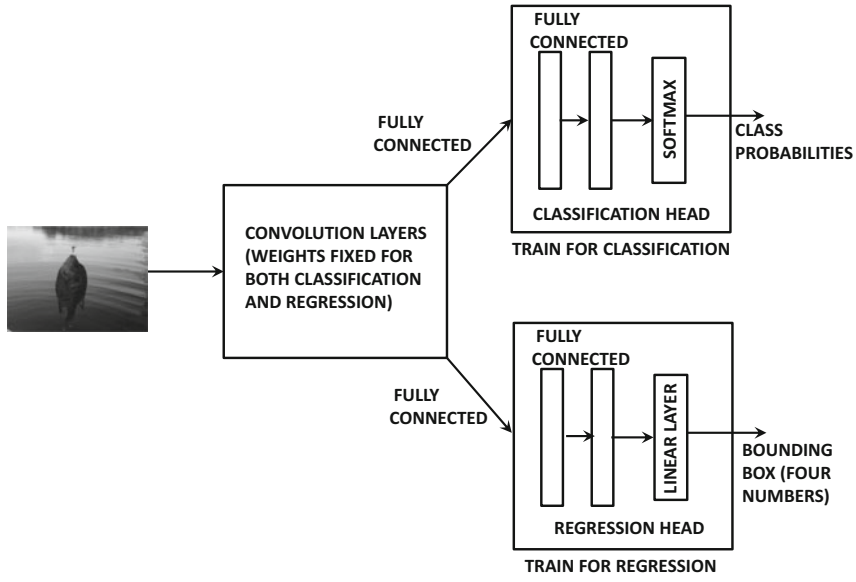


Figure 9.20: The broad framework of classification and localization

two fully connected layers. This is because the semantic nature of the features extracted from the convolution network are often highly generalizable across a wide variety of tasks. Therefore, one can use the following approach:

1. First, we train a neural network classifier like *AlexNet* or use a pretrained version of this classifier. In the first phase, it suffices to train the classifier only with image-class pairs. One can even use an off-the-shelf pretrained version of the classifier, which was trained on *ImageNet*.

2. The last two fully connected layers and softmax layers are removed. This removed set of layers is referred to as the *classification head*. A new set of two fully connected layers and a linear regression layer is attached. Only these layers are then trained with training data containing images and their bounding boxes. This new set of layers is referred to as the *regression head*. Note that the weights of the convolution layers are fixed, and are not changed. Both the classification and regression heads are shown in Figure 9.20. Since the classification and regression heads are not connected to one another in any way, these two layers can be trained independently. The convolution layers play the role of creating visual features for both classification and regression.

3. One can optionally fine-tune the convolution layers to be sensitive to both classification and regression (since they were originally trained only for classification). In such a case, both classification and regression heads are attached, and the training data for images, their classes, and bounding boxes are shown to the network. Backpropagation is used to fine-tune all layers. This full architecture is shown in Figure 9.20.

4. The entire network (with both classification and regression heads attached) is then used on the test images. The outputs of the classification head provide the class probabilities, whereas the outputs of the regression head provide the bounding boxes.

Figure 9.21: Example of object detection. Four objects ("*fish*," "*girl*," "*bucket*," and "*seat*") are identified together with their bounding boxes. The image is illustrative only.

One can obtain better results with a sliding-window approach; the basic idea is to slide a window over the image to identify multiple locations, and then integrate the results of the different runs. An example of this approach is the *Overfeat* method [462]. Refer to the bibliographic notes for pointers to other localization methods.

9.6.3 Object Detection

Object detection is very similar to object localization, except that there is a *variable* number of objects of different classes in the image. The goal is to identify all the objects in the image together with their classes. We have shown an example of object detection in Figure 9.21, in which there are four objects corresponding to the classes "*fish*," "*girl*," "*bucket*," and "*seat*." The bounding boxes of these classes are also shown in the figure. Object detection is generally a more difficult problem than that of localization because of the variable number of outputs. In fact, one does not even know a priori how many objects there are in the image; therefore, one cannot use a fixed number of classification or regression heads.

One option is to use a sliding window. In this approach, one tries all possible bounding boxes in the image, on which the object localization approach is applied to detect a single object. As a result, one might detect different objects in different bounding boxes, or the same object in overlapping bounding boxes. The detections from the different bounding boxes can then be integrated in order to provide the final result. Unfortunately, the approach can be rather expensive. For an image of size $L \times L$, the number of possible bounding boxes is L^4. Note that one would have to perform the classification/regression for each of these L^4 possibilities for each image at testing time. This is a problem, because testing times are always expected to be modest enough to provide real-time responses.

In order to address this issue, *region proposal methods* were advanced. A region proposal method is a general-purpose object detector that identifies promising regions, which are used to first create a set of candidate bounding boxes. Then, the object classification/localization method is run in each of them. Note that some candidate regions might not have valid objects, and others might have overlapping objects. These are then used to integrate and identify all the objects in the image. This broader approach has been used in various techniques like *MCG* [182], *EdgeBoxes* [594], and *SelectiveSearch* [519].

9.6.4 Natural Language and Sequence Learning with TextCNN

While recurrent networks represent the traditional approach for text processing, the use of convolutional neural networks has become increasingly popular. At first sight, convolutional neural networks do not seem like a natural fit for text-mining tasks. First, image shapes are interpreted in the same way, irrespective of where they are in the image. This is not quite the case for text, where the position of a word in a sentence seems to matter quite a bit. Second, issues such as position translation and shift cannot be treated in the same way in text data. Neighboring pixels in an image are usually very similar, whereas neighboring words in text are almost never the same. In spite of these differences, convolutional networks exhibit excellent performance for text and are highly parallelizable (unlike recurrent networks). High parallelizability enables larger training data sets and better accuracy.

Just as an image is represented as a 2-dimensional object with an additional depth dimension defined by the number of color channels, a text sequence is represented as 1-dimensional object with depth defined by its dimensionality of representation. The naive solution of using one-hot encoding is a bad idea, as it blows up the dimensionality of representation (and corresponding parameter footprints). The lexicon size of a typical corpus may often be of the order of 10^6. Therefore, various types of pretrained embeddings of words, such as *word2vec* or *GLoVe* [384] are used (cf. Chapter 3) in lieu of the one-hot encodings of the individual words. Such word encodings also have the advantage of being semantically rich, and the dimensionality of the representation can be reduced to a few thousand (from a hundred-thousand). Therefore, instead of 3-dimensional boxes with a spatial extent and a depth (color channels/feature maps), the filters for text data are 2-dimensional boxes with a window (sequence) length for sliding along the sentence and a depth defined by the *word2vec* embedding. In later layers of the convolutional network, the depth is defined by the number of filters in the previous layer (as in image data). In image data, one performs convolutions at all 2-dimensional locations, whereas in text data one performs convolutions at all 1-dimensional points in the sentence with the same filter. Convolution kernels of different widths may be used to capture correlations between different numbers of words. As in the case of image data, padding may be used to prevent uncontrolled reduction of the output sizes of convolutions, and strides or max-pooling may be used to explicitly control reductions in the temporal footprints of convolutions. In some cases, one might perform the final max-pooling over the entire temporal foot-print, so that each channel produces exactly one output, and there is no sequential ordering among the outputs from different channels. These outputs are then subject to a fully connected set of layers, which further feed into the output layer. One can view the process of max-pooling as that of finding the dominant concepts along each *word2vec* dimension (channel), and then viewing the input sentence as an amalgamation of these concepts. For example, each of the channels of a max-pooling operation will be able to capture when a sentence is about a particular concept (e.g., restaurants, reviews) and if it conveys a positive sentiment. The TextCNN model has been successfully used in *sentiment analysis* applications.

9.6.5 Video Classification

Videos can be considered generalizations of image data in which a temporal component is inherent to a sequence of images. This type of data can be considered *spatio-temporal data*, which requires us to generalize the 2-dimensional spatial convolutions to 3-dimensional spatio-temporal convolutions. Each frame in a video can be considered an image, and one therefore receives a sequence of images in time. Consider a situation in which each image

is of size $224 \times 224 \times 3$, and a total of 10 frames are received. Therefore, the size of the video segment is $224 \times 224 \times 10 \times 3$. Instead of performing spatial convolutions with a 2-dimensional spatial filter (with an additional depth dimension capturing 3 color channels), we perform spatiotemporal convolutions with a 3-dimensional spatiotemporal filter (and a depth dimension capturing the color channels). Here, it is interesting to note that the nature of the filter depends on the data set at hand. A purely sequential data set (e.g., text) requires 1-dimensional convolutions with windows, an image data set requires 2-dimensional convolutions, and a video data set requires 3-dimensional convolutions. We refer to the bibliographic notes for pointers to 3-dimensional convolutions for video classification.

An interesting observation is that 3-dimensional convolutions add only a limited amount to what one can achieve by averaging the classifications of individual frames by image classifiers. A part of the problem is that motion adds only a limited amount to the information that is available in the individual frames for classification purposes. Furthermore, sufficiently large video data sets are hard to come by. For example, even a data set containing a million videos is often not sufficient because the amount of data required for 3-dimensional convolutions is much larger than that required for 2-dimensional convolutions. Finally, 3-dimensional convolutional neural networks are good for relatively short segments of video (e.g., half a second), but they might not be so good for longer videos.

For the case of longer videos, it makes sense to combine recurrent neural networks (or LSTMs) with convolutional neural networks. For example, we can use 2-dimensional convolutions over individual frames, but a recurrent network is used to carry over states from one frame to the next. One can also use 3-dimensional convolutional neural networks over short segments of video, and then hook them up with recurrent units. Such an approach helps in identifying actions over longer time horizons. Refer to the bibliographic notes for pointers to methods that combine convolutional and recurrent neural networks.

9.7 Summary

This chapter discusses the use of convolutional neural networks with a primary focus on image processing. These networks are biologically inspired and are among the earliest success stories of the power of neural networks. An important focus of this chapter is the classification problem, although these methods can be used for additional applications such as unsupervised feature learning, object detection, and localization. Convolutional neural networks typically learn hierarchical features in different layers, where the earlier layers learn primitive shapes, whereas the later layers learn more complex shapes. The backpropagation methods for convolutional neural networks are closely related to the problems of deconvolution and visualization. Convolutional neural networks have also been generalized to text nd sequence processing.

9.8 Bibliographic Notes and Software Resources

The earliest inspiration for convolutional neural networks came from Hubel and Wiesel's experiments with the cat's visual cortex [223]. Based on many of these principles, the notion of the neocognitron was proposed [132]. These ideas were then generalized to the first convolutional network, which was referred to as *LeNet-5* [285]. An early discussion on the best practices and principles of convolutional neural networks may be found in [472]. An excellent overview of convolutional neural networks may be found in [246]. A tutorial on convolution arithmetic is available in [111]. Applications are discussed in [289].

Competitions such as the *ImageNet* challenge (*ILSVRC*) [606] have served as sources of some of the best algorithms over the last five years. Examples of neural networks that have done well at various competitions include *AlexNet* [263], *ZFNet* [581], *VGG* [474], *GoogLeNet* [501], and *ResNet* [194]. The *ResNet* is closely related to highway networks [524], and it provides an iterative view of feature engineering. A useful precursor to *GoogLeNet* was the Network-in-Network (NiN) architecture [306], which illustrated some useful design principles of the inception module (such as the use of bottleneck operations). Several explanations of why *ResNet* works well are provided in [195, 524]. The use of inception modules between skip connections is proposed in [559]. The use of stochastic depth in combination with residual networks is discussed in [221]. Wide residual networks are proposed in [574]. A related architecture, referred to as *FractalNet* [274], uses both short and long paths in the network, but does not use skip connections. Training is done by dropping subpaths in the network, although prediction is done on the full network. Squeeze-and-Excitation networks are discussed in [219]. A recent advancement for image recognition is the use of *capsule networks* [430] to better model hierarchical relationships.

The work in [482] proposes that it makes sense to replace the max-pooling layer with a convolutional layer with increased stride. Not using a max-pooling layer is an advantage in the construction of an autoencoder because one can use a convolutional layer with a fractional stride within the decoder [398]. Fractional strides place zeros within the rows and columns of the input volume, when it is desired to increase the spatial footprint from the convolution operation. The notion of *dilated convolutions* [569] in which zeros are placed within the rows/columns of the filter (instead of input volume) is also sometimes used. The connections between deconvolution networks and gradient-based visualization are discussed in [476, 482]. Simple methods for inverting the features created by a convolutional neural network are discussed in [105]. The work in [320] discuss how to reconstruct an image optimally from a given feature representation. The earliest use the convolutional autoencoder is discussed in [403]. Several variants of the basic autoencoder architecture were proposed in [329, 579, 580]. One can also borrow ideas from restricted Boltzmann machines to perform unsupervised feature learning. One of the earliest such ideas that uses Deep Belief Nets (DBNs) is discussed in [291]. The use of different types of deconvolution, visualization, and reconstruction is discussed in [135, 579–581]. A very large-scale study for unsupervised feature extraction from images is reported in [276].

There are some ways of learning feature representations in an unsupervised way, which seem to work quite well. The work in [79] clusters on small image patches with a k-means algorithm in order to generate features. The centroids of the clusters can be used to extract features. Another option is use random weights as filters in order to extract features [88, 232, 442]; the work [442] shows that a combination of convolution and pooling becomes frequency selective and translation invariant, even with random weights.

A discussion of neural feature engineering for image retrieval is provided in [17]. Numerous methods have been proposed in recent years for image localization. A particularly prominent system in this regard was *Overfeat* [462], which was the winner of the 2013 *ImageNet* competition. This method used a sliding-window approach in order to obtain results of superior quality. Variations of *AlexNet*, *VGG*, and *ResNet* have also done well in the *ImageNet* competition. Some of the earliest methods for object detection were proposed in [90, 122]. The latter is also referred to as the *deformable parts model* [122]. These methods did not use neural networks or deep learning, although some connections have been drawn [170] between deformable parts models and convolutional neural networks. In the deep learning era, numerous methods like *MCG* [182], *EdgeBoxes* [594], and *SelectiveSearch* [519] have been proposed. The main problem with these methods is that they

are somewhat slow. Recently, the *Yolo* method, which is a fast object detection method, was proposed in [407]. The use of convolutional neural networks for image segmentation is discussed in [190]. Region proposal networks are discussed in [412]. Texture synthesis and style transfer methods with convolutional neural networks are proposed in [137, 138, 237]. Tremendous advances have been made in recent years in facial recognition with neural networks. The early work [275, 423] showed how convolutional networks can be used for facial recognition. Deep variants are discussed in [380, 490, 491].

Convolutional neural networks for natural language processing are discussed in [81, 82, 104, 238, 250, 537]. These methods often leverage on *word2vec* or GloVe methods to start with a richer set of features [337, 384]. The notion of recurrent and convolutional neural networks has also been combined for text classification [266]. The use of character-level convolutional networks for text classification is discussed in [585]. Methods for image captioning by combining convolutional and recurrent neural networks are discussed in [236, 529]. The use of convolutional neural network principles for graph-structured data is discussed in the next chapter. A discussion of the use of convolutional neural networks in time-series and speech is provided [282].

Video data can be considered the spatiotemporal generalization of image data from the perspective of convolutional networks [505]. The use of 3-dimensional convolutional neural networks for large-scale video classification is discussed in [18, 233, 244, 517], and the works in [18, 233] proposed the earliest methods for 3-dimensional convolutional neural networks in video classification. All the neural networks for image classification have natural 3-dimensional counterparts. For example, a generalization of *VGG* to the video domain with 3-dimensional convolutional networks is discussed in [517]. Surprisingly, the results from 3-dimensional convolutional networks are only slightly better than single-frame methods, which perform classifications from individual frames of the video. An important observation is that individual frames already contain a lot of information for classification purposes, and the addition of motion often does not help for classification, unless the motion characteristics are essential for distinguishing classes. Video classification also requires lots of training data — even large repositories, such as those in [244] (containing over a million *YouTube* videos) seem to be insufficient for robust training. After all, video processing requires 3-dimensional convolutions that are far more complex than the 2-dimensional convolutions in image processing. As a result, it is often beneficial to combine hand-crafted features with the convolutional neural network [534]. Another useful feature that has found applicability in recent years is the notion of optical flow [53]. The use of 3-dimensional convolutional neural networks is helpful for classification of videos over shorter time scales. Another common idea for video classification is to combine convolutional neural networks with recurrent neural networks [18, 102, 367, 475]. The work in [18] was the earliest method for combining recurrent and convolutional neural networks. The use of recurrent neural networks is helpful when one has to perform the classification over longer time scales. A recent method [22] combines recurrent and convolutional neural networks by making every neuron in the convolution network to be recurrent.

Software Resources and Data Sets

A variety of packages are available for deep learning with convolutional neural networks like *Caffe* [596], *Torch* [597], *Theano* [598], and *TensorFlow* [599]. Extensions of *Caffe* to Python and MATLAB are available.

Off-the-shelf feature extraction methods with pretrained models are discussed in [234, 406]. In cases where the nature of the application is very different from *ImageNet* data, it might make sense to extract features only from the lower layers of the pretrained model (which capture only primitive features). A discussion of feature extraction from *Caffe* may be found in [609]. A "model zoo" of pretrained models from *Caffe* may be found in [610]. *Theano* is Python-based, and it provides high-level packages like *Keras* [600] and *Lasagne* [601] as interfaces. An open-source implementation of convolutional neural networks in MATLAB, referred to as *MatConvNet*, may be found in [522]. The code and parameter files for *AlexNet* are available at [608].

The two most popular data sets for testing convolutional neural networks are *MNIST* and *ImageNet*. Both these data sets are described in detail in Chapter 1. The *MNIST* data set is quite well behaved because its images have been centered and normalized. As a result, the images in *MNIST* can be classified accurately even with conventional machine learning methods, and therefore convolutional neural networks are not necessary. On the other hand, the images in *ImageNet* contain images from different perspectives, and do require convolutional neural networks. Nevertheless, the 1000-category setting of *ImageNet*, together with its large size, makes it a difficult candidate for testing in a computationally efficient way. A more modestly sized data set is *CIFAR-10* [607]. This data set contains only 60,000 instances divided into ten categories, and contains 6,000 color images. Each image in the data set has size $32 \times 32 \times 3$. The *CIFAR-10* data set is often used for smaller scale testing, before a more large-scale training is done with *ImageNet*. The *CIFAR-100* data set is just like the *CIFAR-10* data set, except that it has 100 classes, and each class contains 600 instances. The 100 classes are grouped into 10 super-classes.

9.9 Exercises

1. Consider a 1-dimensional time-series with values 2, 1, 3, 4, 7. Perform a convolution with a 1-dimensional filter 1, 0, 1 and zero padding.

2. For a one-dimensional time series of length L and a filter of size F, what is the length of the output? How much padding would you need to keep the output size to a constant value?

3. Consider an activation volume of size $13 \times 13 \times 64$ and a filter of size $3 \times 3 \times 64$. Discuss whether it is possible to perform convolutions with strides 2, 3, 4, and 5. Justify your answer in each case.

4. Work out the sizes of the spatial convolution layers for each of the columns of Table 9.2. In each case, we start with an input image volume of $224 \times 224 \times 3$.

5. Work out the number of parameters in each spatial layer for column D of Table 9.2.

6. Download an implementation of the *AlexNet* architecture from a neural network library of your choice. Train the network on subsets of varying size from the *ImageNet* data, and plot the top-5 error with data size.

7. Compute the convolution of the input volume in the upper-left corner of Figure 9.2 with the horizontal edge detection filter of Figure 9.1(b). Use a stride of 1 without padding.

8. Perform a 4×4 pooling at stride 1 of the input volume in the upper-left corner of Figure 9.4.

9. Discuss the various type of pretraining that one can use in the image captioning application discussed in section 8.7.2 of Chapter 8.

10. You have ratings of users for different images. Show how you can combine a convolutional neural network with the collaborative filtering ideas discussed in Chapter 3 to create a content-sensitive image recommender system.

11. Show that an average pooling operation on an image with d channels can be implemented as d strided convolution operations with all filter weights *fixed* to 0 or constant a for appropriately chosen a. What is the value of a?

Chapter 10

Graph Neural Networks

"Everyone should build their network before they need it." — Dave Delaney

10.1 Introduction

Graphs are used in a wide variety of application-centric settings, such as the Web, social networks, communication networks, and chemical compounds. Graphs can be either *directed* or *undirected*. In directed graphs, edges have direction, whereas in undirected graphs, the edges do not have direction. The edges in directed graphs are represented by lines with arrowheads, whereas edges in undirected graphs are represented by lines without arrowheads. Some examples of real-world graphs are as follows:

1. In a social network, a node is a user, and an edge is a friendship link between the users. Friendship links are undirected, whereas follower-followee links are directed.
2. In a Web network, a Web page is a node, and a hyperlink is a directed edge.
3. In a chemical compound, an element is a node and a bond is an edge. This setting is slightly different from the first two settings, as we typically have multiple small graphs rather than a single large graph for learning purposes.

Graphs are typically represented with the use of *adjacency matrices*. For a graph with n nodes, its adjacency matrix A is an $n \times n$ matrix in which the (i, j)th entry is given by the weight of the directed edge from node i to node j. In the event that the edges are not directed, the adjacency matrix is symmetric. Most of this chapter works with the assumption of undirected networks. For undirected networks, the set of nodes incident on node i is denoted by $A(i)$. In some cases, it is assumed that the entries of the adjacency matrix are drawn from $\{0, 1\}$, when edge weights are not available.

There are two distinct settings in which graph neural networks are used. In the first setting, a single large graph (e.g., *Twitter* network) is available with features on the nodes. These node features are then distilled into a refined latent space on the basis of structural

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_10

locality (so that connected nodes have similar representations). The node representations are used for node-centric prediction. In the second setting, a large collection of smaller graphs (e.g., chemical compounds) is available, and it is desired to embed each *graph* (rather than node) into a latent space. The graph representations are used for graph-centric prediction. The second setting is significantly more challenging than the first, since one must generalize the structure-feature learning *across* multiple graphs.

Chapter Organization

This chapter is organized as follows. In the next section, we review some simple neural networks for node embedding based on the discussions in Chapter 3. A specific graph neural network architecture for node embedding that is optimized to the graph setting is introduced in section 10.3. The backpropagation algorithm for graph neural networks is discussed in section 10.4. The embedding of full graphs is discussed in section 10.5. Applications of graph neural networks are discussed in section 10.6. A summary is given in section 10.7.

10.2 Node Embeddings with Conventional Architectures

In this section, we start with reviewing some simple neural networks for node embedding based on an earlier discussion in section 3.7 of Chapter 3. Consider an $n \times n$ adjacency matrix $A = [a_{ij}]$ for a graph with n nodes. The entry a_{ij} is 1 if an undirected edge exists between nodes i and j. Furthermore, the matrix A is symmetric, because we have $a_{ij} = a_{ji}$ for an undirected graph. As discussed in section 3.7, one can use logistic matrix factorization in order to generate the entries of the matrix A. Each entry of A is generated using the matrix of Bernoulli parameters in $f(UV^T)$, where $f(\cdot)$ is the element-wise application of the sigmoid function to each entry x of the matrix:

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{10.1}$$

Therefore, if $\overline{u}_i$ is the ith row of U and $\overline{v}_j$ is the jth row of V, we have the following:

$$a_{ij} \sim \text{Bernoulli distribution with parameter } f(\overline{u}_i \cdot \overline{v}_j) \tag{10.2}$$

The neural architecture of the model is shown in Figure 10.1 for a toy graph containing 5 nodes. The input is the one-hot encoded index of a row in A (i.e., node), and the output is the list of all 0/1 values for all nodes in the network. In this particular case, we have shown the input for node 3 and its corresponding output. Since the node 3 has three neighbors, the output vector contains three 1s. The outputs correspond to sigmoid units with log-likelihood loss. It is possible to increase the number of hidden layers in order to create a more expressive model.

Most networks are sparse, and therefore the number of 0s in the output is much greater than the number of 1s. Therefore, it is possible to use a similar trick as the SGNS model of *word2vec* and drop many of the 0s with the use of negative sampling. Each negative node

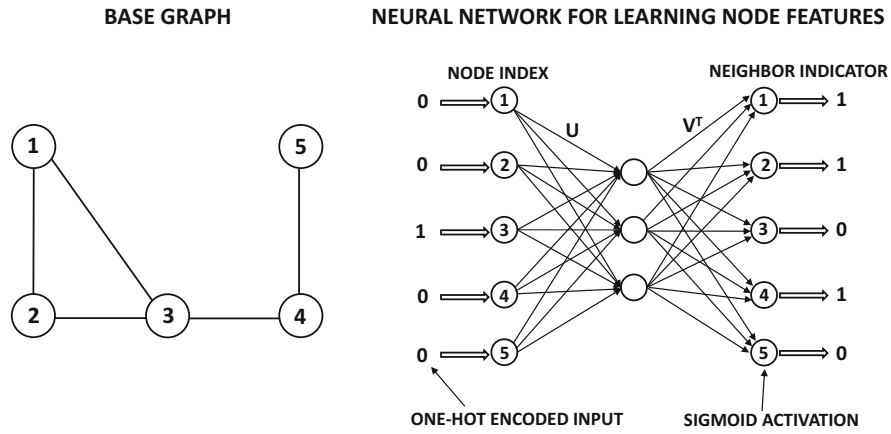


Figure 10.1: Revisiting Figure 3.16: A graph of five nodes is shown together with a neural architecture for row index to neighbor indicator mapping. The shown input and output represent node 3 and its neighbors.

in the output can be sampled as a sublinear power of its node degree. The backpropagation is performed only from the sampled nodes in order to perform the embedding while saving computational time. In fact, the architecture is very similar to that of *word2vec*, which has been shown to be equivalent to logistic matrix factorization in Chapter 3. Therefore, this approach is equivalent to logistic matrix factorization of the adjacency matrix with some additional sampling improvements (for computational performance).

In most practical applications, adjacency matrices are not binary because edges have weights. As discussed in section 3.7, one can handle weighted edges with the use of sampling, where the outputs are sampled binary values and the probability of sampling is proportional to the weight. Absent edges (for a particular input node) can also be sampled in proportional to a sublinear power of the weighted node degree at the other end of the input node. At its core, this class of methods closely mimics *word2vec* by implementing neural matrix factorization.

The aforementioned model performs unsupervised node embedding, although one can extend this approach to supervised settings. For example, in the *collective classification problem*, a subset of the nodes are labeled with one of m classes indexed by $\{1, \ldots, m\}$. The goal of the collective classification problem is to assign class labels to all the unlabeled nodes in the graph. The architecture of Figure 10.1 can be modified as follows. A single softmax output layer with m possible outputs is added to the architecture. This softmax layer is attached to all the hidden units (not shown in Figure 10.1). Note that the additional output layer does not replace the outputs included in Figure 10.1 but it *adds* to these outputs (thereby creating a *semisupervised* model). This softmax unit outputs the probability of each class label. The loss function also needs to be modified since we have two sets of outputs. In addition to the log-loss function of the logistic outputs corresponding to the presence or absence of different nodes, the cross-entropy loss is added with respect to the observed class label in the softmax output. The two components of the loss can be weighted appropriately to reach a desired degree of supervision. For nodes in which the output labels are missing, backpropagation is performed only from the unsupervised output units (corresponding to node identifiers). Although this architecture does not use input node embeddings in the

neural architecture, it is relatively easy to enhance this model with additional input/output nodes to incorporate input node features (see Exercise 1).

10.2.1 Adjacency Matrix Representation and Feature Engineering

The neural network architecture of Figure 10.1 uses the *immediate* connectivity between nodes to create input-output pairs. In practice, the feature representation of a node should depend not only on its immediate neighbors, but also on nodes that are connected by short paths. One way of achieving this goal is to change the input edge weights with the use of random walks — the weight of an edge between two nodes is modified to be the number of times that one node occurs in a short random walk starting at the other node, and vice versa. The number of random walks of length k between nodes i and j are contained in the (i, j)th entry of A^k. Therefore, one can create a new weighted adjacency matrix with the use of the Katz measure, in which the number of random walks of all possible lengths are aggregated after decaying them by length:

$$A_\alpha = \sum_{i=1}^{\infty} (\alpha\, A)^i = (I - \alpha A)^{-1} - I$$

Here, α is a decay parameter that lies in the range $(0, 1)$. Using $\alpha < 1$ is essential in order to ensure that the infinite summation converges, and α should always be chosen to be less than the inverse of the magnitude of the largest eigenvector of A. The Katz measure is essentially the sum of the number of all decay weighted walks between a pair of nodes, where longer walks are de-emphasized by the decay factor. Using the walk-centric adjacency matrix A_α instead of A leads to more robust results because indirect connectivity is incorporated in the input representation.

One can view this approach as a form of feature engineering in order to improve the *input representation* of the adjacency matrix. The feature engineering compensates for the shallow nature of the underlying neural network — after all, the primary purpose of depth is to engineer increasingly sophisticated features with each layer of depth. Combining this type of handcrafted feature engineering with a shallow network goes against the spirit of neural network design, which is generally intended to be an end-to-end system that can perform the feature engineering automatically. To achieve this goal, graph neural networks directly use the structure of the graph in the learning process across multiple layers (just like convolutional networks).

10.3 Graph Neural Networks: The General Framework

Graph neural network enforce similarity in the representations of adjacent nodes, just as convolutional neural networks leverage the idea of representational similarity in adjacent pixels. These networks use a layered architecture in which a one-to-one correspondence exists between neural network nodes and graph nodes in each layer. The node representations in a given layer are obtained as a function of the representations of that node and its neighboring nodes in the previous layer. This type of adjacency is also leveraged in convolutional neural networks for images (with the use of filters). By repeatedly performing this type of adjacency-based transformations over a small number of layers, the embeddings of nodes are able to incorporate the structure of their locality. As a result, the network connectivity

structure is leveraged to promote similar representations for highly connected nodes. The transmission of hidden values between nodes across time stamps in graph neural networks is referred to as *message passing*.

For the purpose of this section, we will assume that we have a graph containing n nodes, whose input feature vectors are $\overline{x}_1 \ldots \overline{x}_n$. The feature vector for the ith node is the d-dimensional column vector $\overline{x}_i$. In practice, input feature vectors for graphs may contain both application-specific features (e.g., user profile in a social network) and hand-crafted features based on the local structure of the graph (e.g., node degree and neighborhood size). Hand-crafted features offer a high degree of flexibility because one could technically include the entire row of the adjacency matrix for node i within $\overline{x}_i$. The main problem with using such an approach is that it makes the input sparse and high-dimensional. Some methods use the dominant eigenvectors of the normalized adjacency matrix[1] in order to first generate a low-dimensional embedding of each node. This low-dimensional embedding is concatenated with various application-specific and structural features in order to create the input $\overline{x}_i$ for the ith node. Most models omit the use of the adjacency matrix entirely within the input, and use it only to define the adjacency-based transformations in the neural network.

For simplicity, we first assume that we are always working with binary adjacency matrices and generalize the discussion later to weighted adjacency matrices. Although most of the discussion in this chapter is designed for undirected graphs, we will first start with an example of a directed graph. The reason that we start with directed graphs is that edge direction causes some additional complexities that do not occur in the case of undirected graphs. After briefly discussing how these complexities can be addressed in general, we will devote most of the chapter to a simplified discussion of undirected graphs.

We now introduce the notations for the hidden states in the neural architecture. The input features are assumed to be the "hidden features" at the zeroth layer. Therefore, the (column vector) inputs, $\overline{x}_1 \ldots \overline{x}_n$, can also be denoted by the "embeddings" $\overline{h}_1^{(0)}, \overline{h}_2^{(0)}, \ldots \overline{h}_n^{(0)}$. The representation of the ith node in the kth layer is denoted by $\overline{h}_i^{(k)}$, where k varies from the input layer index 0 to the output layer index T. The superscript '(k)' should not be confused with an exponent. Like $\overline{x}_i$, each $\overline{h}_i^{(k)}$ is a column vector. Note that a one-to-one correspondence exists between graph nodes and neural network nodes *in each layer*, resulting in n layer-specific nodes $\overline{h}_1^{(k)}, \overline{h}_2^{(k)}, \ldots \overline{h}_n^{(k)}$. The hidden representation in the kth layer is assumed to be p_k-dimensional, which can vary with the layer index. The value of p_0 is therefore equal to the input dimensionality d of the node features.

[1]For an adjacency matrix A, we can first divide the weight of the edge with the geometric mean of the weighted node degrees at its end points in order to normalize it to A_c. Then, the symmetric matrix $A_c A_c^T$ is diagonalized as $P\Delta P^T$, and the first s columns of P corresponding to the largest eigenvalues are retained in order to create an $n \times s$ matrix $P_s\Delta_s$, where Δ_s is the corresponding $s \times s$ diagonal matrix of eigenvalues. The value of s is typically orders of magnitude less than n. The ith row of P_s contains an embedding of the ith node. Therefore, the ith row of P_s can be input as a structural feature to the neural network input node for the ith graph node.

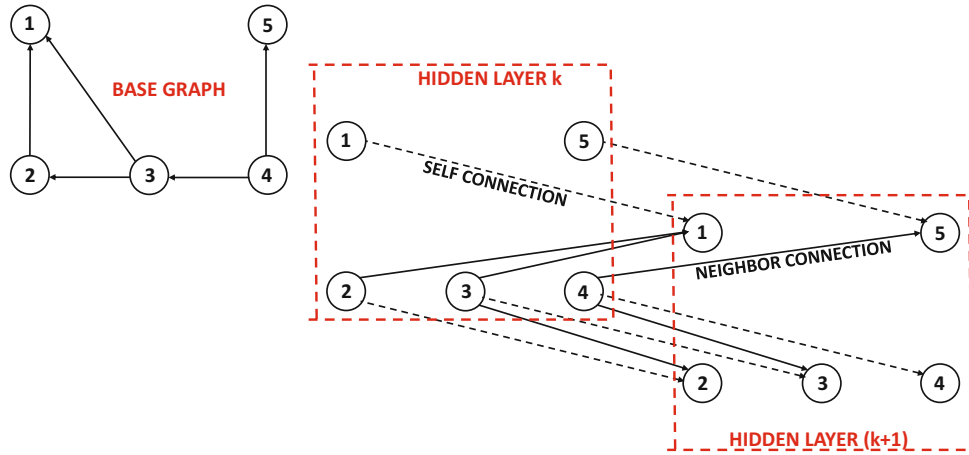


Figure 10.2: A directed graph of five nodes is shown together with an illustration of how it is converted to a layered architecture with both self connections and neighbor connections.

The connections in the neural network are also inherited from the structure of the graph. There are two types of connections in the neural network between a pair of adjacent layers, corresponding to *self connections* and *neighbor connections*. The two types of connections are defined as follows:

1. *Self connections:* Each node $\overline{h}_j^{(k)}$ is directly connected to its corresponding node $\overline{h}_j^{(k+1)}$ in the next layer. The number of self connections between a pair of layers is exactly equal to the number of nodes in the original graph. The purpose of the self connections is to inherit some characteristics of the embedding from one time stamp to the next.

2. *Neighbor connections:* For each edge (i, j) in the original graph, a corresponding edge exists in the graph from the layer k to the layer $(k + 1)$. Therefore, an edge exists from the neural network node containing the hidden representation $\overline{h}_i^{(k)}$ to the node containing $\overline{h}_j^{(k+1)}$. The number of neighbor connections between a pair of layers is exactly equal to the number of edges in the original graph. Since the embeddings of connected nodes are expected to be related, it makes sense for the embedding of a node to made dependent on those of its incident nodes. Here, a key point is that arises is from the direction of the edges. For example, since there is an edge from node 4 to node 5 in the base graph of Figure 10.2, and there is an edge from node 4 of layer k to node 5 of layer-$(k + 1)$. Such an approach will learn node embeddings based on *indegree connections*. One can also construct a neural network with edge from node 5 of layer k to node 4 of layer $(k+1)$. Such a neural network will learn node embeddings based on *outdegree connections*. By concatenating the two types of features, one can obtain the full embedding.

An example of how a graph may be converted into the structure of a graph neural network is shown in Figure 10.2. The connections between layers k and $(k + 1)$ are shown for a particular base graph. Since there are a total of five nodes in the base graph, there are five self connections between the corresponding nodes of the neural network in adjacent layers. These connections are shown by dashed lines. Furthermore, since there are five edges, there are also a total of five neighbor connections, which are shown by solid lines.

These connections are used to ensure that the hidden state of node i in layer k is generated from that of the same node and its neighbors in the previous layer as follows:

$$(\forall i, k): \quad \overline{h}(k)_i = \text{Function of} \overline{h}(k-1)_i \text{ and all } \overline{h}(k-1)_j \text{ for } j \text{ incident on } i$$

By varying on the choice of this function, one can design a wide variety of graph neural networks with somewhat different properties. The choice of function and its impact on the embedding will be a topic of significant discussion in subsequent sections.

It is also noteworthy that Figure 10.2 assume that neural network connections from one layer to the next obey the direction of the edge in the base (directed) graph. Therefore, the nodes in layer-k only contain outgoing edges of the base graph, and the nodes in layer-$(k+1)$ only contain the incoming edges of the base graph. This does not need to be true in general, as the outgoing edges from a node provide different insights as the incoming edges. In order to understand why this is the case, note that the followers of a rock star on Twitter will be fundamentally different from the (more exclusive) set that the rock star might follow. As a result, the nature of the representation learning will be fundamentally different in the two cases. Therefore, it is possible to modify Figure 10.2 to also include additional edges between layer k and layer $(k+1)$, which are in the reverse direction as the base graph. Therefore, one would have both forward neighbor connections and reverse neighbor connections in the base graph, and the function above needs to treat them differently. In undirected graphs, this distinction does not matter, since all edges are symmetric between two nodes (and therefore both directions of edges are automatically present in the layered network). Furthermore, one does not need to explicitly distinguish between forward and reverse connections. Henceforth, we will assume that all graphs discussed in this chapter are undirected, and the adjacency list of node i is denoted by $A(i)$. Making the assumption of undirected graphs frees us from the challenges of having to separately deal with the indegree and outdegree aspects of the feature embedding.

The structure of the output layer can vary quite a lot. While it is possible to feed only the final hidden layer of the neural network into an output layer with the appropriate output dimensionality, one often also connects the earlier hidden layers to the output layer as well. These types of connections are like skip connections that take advantage of the learning in earlier layers of the neural network. The output for each node can be either unsupervised or supervised, depending on the application at hand. In particular, one can choose one of the following types of outputs:

1. *Unsupervised:* In this case, the output dimensionality for each node is the same as the input dimensionality, and reconstruction loss is used. This is similar to how an autoencoder reconstructs the outputs for embedding multidimensional data.

2. *Supervised:* In this case, *only labeled nodes have trainable outputs for backpropagation purposes although all nodes have outputs for prediction purposes.* For each node in the output layer, we have a neural network node corresponding to an m-way softmax layer, where m is the number of classes. The loss used is the sum of the cross-entropy losses of labeled nodes (with respect to the ground-truth class label). At prediction time, the softmax outputs of the unlabeled nodes are used to predict the probability of each label for a given node.

3. *Semisupervised:* In this case, we have two sets of output nodes — one set performs unsupervised reconstruction for each node, and the other performs the supervised task of predicting the class label with the use of the softmax layer. These two types of nodes have already been discussed above and they are both connected to the final

hidden layer. The overall loss is the weighted sum of the unsupervised and supervised losses. The relative weights of the two losses regulate the degree of supervision, and it can be tuned on a validation set of held-out labeled nodes to maximize classification accuracy.

For supervised and semi-supervised versions of graph neural networks, the main output is the prediction of the class label for unlabeled nodes. For unsupervised and semi-supervised versions of graph neural networks, the hidden variable values of each node in the various hidden layers provides an embedding, which can be used in the context of a wide variety of applications. The embeddings from earlier time layers (i.e., layers closer to the input layer) tend to encode more local and primitive structural features, whereas the embeddings from later time layers tend to encode more global and complex features of the nodes. Both are useful in different types of applications, and are often concatenated to create the full embedding. Next, we will discuss some key functions that are useful for propagating the values in the hidden layers.

10.3.1 The Neighborhood Function

The simplest function used for designing a graph neural network is the neighborhood function. In the neighborhood function, the average embedding of all adjacent nodes is used in order to generate the embedding in layer k from that in layer $(k-1)$. The neighborhood function is defined as follows:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(W^{(k)} \sum_{j \in A(i)} \frac{\overline{h}_j^{(k-1)}}{|A(i)|} + B^{(k)} \overline{h}_i^{(k-1)}\right) \tag{10.3}$$

Here, $\Phi(\cdot)$ is a nonlinear activation function (e.g., ReLU) that is applied in elementwise fashion to its vector argument; $W^{(k)}$ and $B^{(k)}$ are the trainable matrices that are shared across different nodes, and are learned during backpropagation. Both matrices are of size $p_k \times p_{k-1}$. One can view $W^{(k)}$ as a weight matrix that applies to neighborhood connections, and $B^{(k)}$ as a bias matrix that applies to self connections. The trainable matrices $W^{(k)}$ and $B^{(k)}$ are shared across nodes in a layer but they are not shared across different layers. This is similar to a convolutional neural network, which shares the filter across the entire image, but not across different layers.

It is noteworthy that the neighborhood function *averages* the neighborhood embedding rather than *summing* them. This is important for normalization purposes, since most real-world graphs satisfy *power-law degree distributions* [3], which causes the nodes to have vastly different degrees. Using the summation instead of averaging will result in considerable difficulty in generating comparable embeddings for nodes of vastly different degrees. As we will see later, different types of function use different forms of normalization in order to account for the vastly varying node degrees.

The value of each p_k is often smaller than the square root of the number of nodes, and therefore the number of parameters in the $p_k \times p_{k-1}$ matrices $W^{(k)}$ and $B^{(k)}$ are smaller than the number of nodes. The reduced number of parameters is helpful in avoiding overfitting.

10.3.2 Graph Convolution Function

Graph convolutional networks use a similar principle as neighborhood-based aggregation, except that the nature of the underlying degree normalization is different. While the neigh-

borhood aggregation operator uses node-wise normalization with degrees, the graph convolutional operator uses edgewise normalization of the embedding with the geometric mean of the degrees at the two ends of the edge. Furthermore, a separate matrix $B^{(k)}$ is not used for transmitting embedding values across self connections, but a self-loop with weight equal to the node degree is added to the graph. By doing so, one can use a single matrix $W^{(k)}$ to handle both self connections and neighbor connections. Therefore, the graph convolutional function may be written as follows:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(W^{(k)} \sum_{j \in A(i) \cup \{i\}} \frac{\overline{h}_j^{(k-1)}}{\sqrt{|A(i)| \cdot |A(j)|}}\right) \tag{10.4}$$

Note that self connections are included within the above transformation, since the aggregation operation for a node includes its representation from the previous layer. The activation function $\Phi(\cdot)$ is applied in elementwise fashion to its vector argument. By using this type of normalization, embeddings of high-degree nodes have less influence on their neighbors, although high-degree nodes tend to have embeddings of somewhat larger (absolute) components.

The convolutional operation can be expressed in matrix form. The p_k-dimensional hidden representations of the n nodes can be stacked up into an $n \times p_k$ matrix $H^{(k)}$ as follows:

$$H^{(k)} = [\overline{h}_1^{(k)}, \overline{h}_2^{(k)}, \ldots, \overline{h}_n^{(k)}]^T$$

In other words, the ith row of $H^{(k)}$ contains $\overline{h}_i^{(k)}$. Let Δ be the $n \times n$ diagonal degree matrix, which contains the sum of the row elements of the (symmetric) adjacency matrix A. The $n \times n$ *normalized and conditioned* adjacency matrix A_c is defined as follows:

$$A_c = \Delta^{-1/2}(A + \Delta)\Delta^{-1/2} = I + \Delta^{-1/2} A \Delta^{-1/2}$$

The above operation normalizes each edge weight in A with the geometric mean of node degrees at its ends, and adds 1 to each diagonal element (thereby flipping its value from 0 to 1). The 1s on the diagonal are important for performing operations across self connections. Then, the updates may be stated[2] in matrix form as follows:

$$H^{(k)} = \Phi\left(A_c H^{(k-1)} W^{(k)}\right) \tag{10.5}$$

The activation function $\Phi(\cdot)$ is applied in elementwise fashion to its matrix argument. Since the largest matrix A_c in the above product is sparse, the multiplication can be performed efficiently.

10.3.3 GraphSAGE

GraphSAGE is a generalized variant of the neighborhood-based aggregation function, which keeps the operations along self connections and neighbor connections separate. We restate

[2]Strictly speaking, the $p_{k-1} \times p_k$ matrix $W^{(k)}$ in Equation 10.5 is the transposed form of the $p_k \times p_{k-1}$ matrix $W^{(k)}$ in the vector-wise updates of Equation 10.4. We have chosen this type of overloaded notation in order to retain consistency with the broader research literature on neural networks. This observation is significant from the implementation perspective, because one has to initialize $W^{(k)}$ to be of size either $p_k \times p_{k-1}$ or of size $p_{k-1} \times p_k$ depending on whether one codes up Equation 10.4 or 10.5.

the neighborhood-based aggregation function of Equation 10.3:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(W^{(k)} \sum_{j \in A(i)} \frac{\overline{h}_j^{(k-1)}}{|A(i)|} + B^{(k)} \overline{h}_i^{(k-1)}\right)$$

Note that the self embeddings of the previous layer are *aggregated* with the neighbor embeddings in the neighborhood model. Instead, we might choose to *concatenate* them (i.e., stack them into a longer vector) as follows:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left[\begin{array}{c} W^{(k)} \sum_{j \in A(i)} \frac{\overline{h}_j^{(k-1)}}{|A(i)|} \\ B^{(k)} \overline{h}_i^{(k-1)} \end{array}\right]$$

The function $\Phi(\cdot)$ is a nonlinear activation function that is applied in elementwise fashion to its vector argument. What is the advantage of concatenation over aggregation? By concatenating, the features across self connections are emphasized, as they are not "lost" by successive aggregation with other neighborhood features. As we will see subsequently in this chapter, successive aggregation often leads to the feature representations of all nodes becoming very similar. This problem is referred to as *over-smoothing* [300], which is mitigated to some extent by concatenation.

GraphSAGE can be formulated in a more general way by replacing the neighborhood aggregation in the forward pass updates with a general aggregation operator $\Gamma(\cdot)$:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left[\begin{array}{c} W^{(k)} \Gamma(\{\overline{h}_j^{(k-1)} : \forall j \in A(i)\}) \\ B^{(k)} \overline{h}_i^{(k-1)} \end{array}\right] \tag{10.6}$$

Note that Equation 10.3.3 can be obtained as a special case of Equation 10.6 by setting the operator $\Gamma(\cdot)$ as follows:

$$\Gamma(\{\overline{h}_j^{(k-1)} : \forall j \in A(i)\}) = \sum_{j \in A(i)} \frac{\overline{h}_j^{(k-1)}}{|A(i)|}$$

However, one can choose other forms of the aggregation operator, as long as it is differentiable (for backpropagation to work) and permutation invariant on the neighbors of a given node. One such important operator is the max-pooling operator:

$$\Gamma(\{\overline{h}_j^{(k-1)} : \forall j \in A(i)\}) = \text{ELEMENTWISE-MAX}_{j \in A(i)} W_{pool} \overline{h}_j^{(k-1)}$$

Here, we are using an additional matrix W_{pool} for pooling, which needs to be learned during backpropagation. Multiplying the hidden representation of each neighbor of the jth node with W_{pool} results in one vector for each of the neighbors. The elementwise maximum over these different vectors is created in order to consolidate (i.e., aggregate) the result into a single vector. The original GraphSAGE publication indicates that the pooling operator works better than the simple aggregation operator.

Another aggregation operator is to use the embedding obtained after applying an LSTM on the neighbor nodes. However LSTMs work on sequences, whereas the neighbors must be treated in a permutation invariant fashion. Therefore, the LSTM is applied on a random permutation of the nodes. The pooling operator and the LSTM operator are similar in

accuracy, although the LSTM operator is much slower. This suggests that the pooling operator is the preferred choice among the operators introduced in this section.

Even though we have presented GraphSAGE with the use of all the neighbor nodes in the aggregation operator, it suffices to use fixed size random samples of the nodes for creating the aggregation operator. Sampling the nodes also makes the updates much more efficient. The approach works best with a small number (two or three) hidden layers.

10.3.4 Handling Edge Weights

The implicit assumption in the discussion so far is that the edges are binary in nature, because all neighbors of a node are treated in uniform fashion. In practice, however, edges do have weights associated with them, and edges with larger weight should be given more importance during the embedding process. Therefore, all neighborhood aggregations are weighted by the corresponding edge weights. Let the weight of edge (i, j) be denoted by c_{ij}. The simple neighborhood aggregation operator of Equation 10.3 may be modified with these edge weights as follows:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(W^{(k)} \sum_{j \in A(i)} \frac{c_{ij}\overline{h}_j^{(k-1)}}{\sum_{j \in A(i)} c_{ij}} + B^{(k)}\overline{h}_i^{(k-1)}\right)$$

The modifications to the updates for the graph convolutional operator are very similar. In fact, the updates in matrix form remain exactly the same, except that the matrices are defined in a weighted way. We restate the matrix-centric updates in graph convolutional networks (cf. Equation 10.5) with the use of the normalized and conditioned adjacency matrix A_c:

$$H^{(k)} = \Phi\left(A_c H^{(k-1)} W^{(k)}\right)$$

The main difference from the unweighted version of the update is in terms of how the matrix A_c is defined. First, the original adjacency matrix A is weighted, and the degree matrix Δ is also the diagonal matrix containing the weighted degrees of the nodes. Then, A_c is defined as $I + \Delta^{-1/2} A \Delta^{-1/2}$.

One can also handle weighted networks by sampling neighbors in proportion to their weights during aggregation. This approach can work quite naturally with algorithms like GraphSAGE, which have some form of sampling already built into the learning process.

10.3.5 Handling New Vertices

Many real world social networks are incremental in which new nodes (which are connected to some of the older nodes in the network) arrive over time. An important property of the shallow networks discussed in section 10.2 (which form the basis of networks like *node2vec* and *DeepWalk*) is that it is not possible to embed new nodes from the model that has already been trained. This is because the input is a one-hot encoding of as many outcomes as the number of nodes. Once the network has been trained with this type of input structure, it is no longer possible to embed new nodes, unless an additional input and output is incorporated, and some updating is performed on the weights with updated inputs. The inputs can change quite drastically with addition of new nodes, especially if random walk measures are used in order to define edge weights. One can use the original weights as a starting point (with the additional input and output added) with zero weights on new

connections in the network. Even though this kind of incremental training is less onerous than performing the entire training from scratch, it can still be quite expensive.

However, the layered networks of this section can be used to embed new nodes even without any additional or incremental training, because the weight matrices are shared across nodes. Therefore, one can apply the convolution operation to the input representation of the new node with the matrix that has already been learned. This procedure can be applied repeatedly across layers to create the output embeddings of new nodes without additional training.

10.3.6 Handling Relational Networks

In relational networks, the edges may be of different types, representing different types of relations between entities. Such networks are also referred to as *heterogeneous information networks*, and they are common in knowledge graphs. For example, a network consisting of actors, directors, and movies might contain various types of edges such as "*directed in*" or "*acted in*" edges. For the purpose of discussion, we assume that there are a total of l different edge types. Then, different weight matrices (for neighborhood aggregation operations in the neural network) are learned corresponding to the various edge types, and the effects of the different edge types are aggregated. Let the weight matrix for the rth edge type in the kth layer be denoted by $W^{(k,r)}$ and $B^{(k,r)}$ in the case of the neighborhood aggregation operator. Furthermore, let $A^1 \ldots A^l$ be the adjacency matrices for the l different edge types, and let $A^1(i) \ldots {}^l(i)$ be the adjacency lists of node i for the l different edge types. Then, the simple neighborhood aggregation updates of Equation 10.3 may be modified as follows:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(\sum_{r=1}^{l} W^{(k,r)} \sum_{j \in A^r(i)} \frac{\overline{h}_j^{(k-1)}}{|A^r(i)|} + \sum_{r=1}^{l} B^{(k,r)} \overline{h}_i^{(k-1)}\right)$$

One can also provide different weights to the different modalities by adding trainable parameters $\beta_1, \beta_2, \ldots \beta_l$ to the different link types:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(\sum_{r=1}^{l} \beta_r W^{(k,r)} \sum_{j \in A^r(i)} \frac{\overline{h}_j^{(k-1)}}{|A^r(i)|} + \sum_{r=1}^{l} B^{(k,r)} \overline{h}_i^{(k-1)}\right)$$

The convolutional updates of Equation 10.5 can also be modified to work with different edge types as follows:

$$H^{(k)} = \Phi\left(\sum_{r=1}^{l} A_c^r H^{(k-1)} W^{(k,r)}\right)$$

Here, the normalized matrix A_c^r is constructed using the adjacency matrix A^r. The degree matrix for each link type is constructed using its own adjacency matrix for normalization purposes.

One can also make the importance of different link types flexible, but using trainable parameters $\beta_1 \ldots \beta_l$:

$$H^{(k)} = \Phi\left(\sum_{r=1}^{l} \beta_r \, A_c^r H^{(k-1)} W^{(k,r)}\right)$$

It is noteworthy that in most practical settings, the edge types (e.g., "*acted-in*") implicitly define the node types (e.g., "*actor*", "*movie*") at the two ends of the edge, and the number

and types of attributes at different node types may be different. In such cases, the different edge-specific weight matrices will have varying sizes in order to accommodate the varying cardinalities of attributes at the two ends.

10.3.7 Directed Graphs

Directed graphs present a somewhat different challenge than undirected graphs, because the effect of incoming nodes on features is distinctively different from that of outgoing nodes. For example, the characteristics of a Twitter user following many other users are different from the characteristics of a user that is followed by many users. Therefore, separate matrices W_{in}, W_{ou}, B_{in}, and B_{ou} are used for incoming and outgoing nodes. Similarly, the incoming incident nodes at node i are denoted by $A_{in}(i)$, whereas the outgoing incident nodes at node i are denoted by $A_{ou}(i)$. It is possible to create separate neighborhood models for the incoming nodes and outgoing nodes and then aggregate them into a single model. The neighborhood model of Equation 10.3 is modified as follows:

$$(\forall i, k): \quad \overline{h}_{in}[i,k] = \Phi\left(W_{in}^{(k)} \sum_{j \in A_{in}(i)} \frac{\overline{h}_j^{(k-1)}}{|A_{in}(i)|} + B_{in}^{(k)} \overline{h}_i^{(k-1)}\right) \quad \text{[Incoming]}$$

$$(\forall i, k): \quad \overline{h}_{ou}[i,k] = \Phi\left(W_{ou}^{(k)} \sum_{j \in A_{ou}(i)} \frac{\overline{h}_j^{(k-1)}}{|A_{ou}(i)|} + B_{ou}^{(k)} \overline{h}_i^{(k-1)}\right) \quad \text{[Outgoing]}$$

$$(\forall i, k): h_i^{(k)} = h_{in}[i,k] + h_{ou}[i,k] \quad \text{[Aggregate]}$$

The above generalization from undirected to directed graphs has been presented for the simplest (neighborhood) model. However, similar changes can also be presented for other models like graph convolutional networks.

10.3.8 Gated Graph Neural Networks

While graph neural network attempt to learn deeper structural properties of the graph by using multiple hidden layers (unlike the unlayered frameworks discussed in section 10.2), the number of layers used is quite limited. In practice, most graph convolution networks use no more than 3 or 4 layers. Like any deep architecture, such networks are prone to the vanishing and exploding gradient problems. More importantly, repeated aggregation of the representations of the nodes in a given neighborhood leads to *over-smoothing*, where all node representations become very similar [300]. While concatenation mitigates the smoothing problem to some extent (as in GraphSAGE), the smoothing problem continues to manifest itself in the aggregated portions of the features; furthermore, the vanishing and exploding gradient problems can occur in the features learned across self connections. This problem is unique to graph neural networks, where nodes are aggregated *before* multiplying with a parameter matrix (unlike other neural networks were representations are multiplied with parameters before aggregating them). Furthermore, all nodes share the same parameter matrix, and the multiplication of each aggregated representation with a parameter matrix is unable to distinguish the underlying representations. Real-world social graphs have the *small-world* property, wherein most pair of nodes are located at a distance of about 3 or 4 hops. As a result, after 3 or 4 layers, all nodes would have contributions from most other nodes, albeit in a somewhat different proportion. This makes it increasingly difficult to distinguish the node representations from one another, as the number of layers increases.

In order to address the vanishing/exploding gradient problems and the over-smoothing problem, ideas from gating (in recurrent neural networks) have been incorporated in order to create deeper networks. Gated recurrent units use similar ideas to the Gated Recurrent Units (GRUs) leveraged in sequence data. Recall that GRUs are able to avoid the vanishing and exploding gradient problems with the use of gated units in the context of a recurrent neural network. A similar idea is used to construct gated graph neural networks.

In order to reduce the number of parameters, gated graph neural networks share the matrix used in conjunction with the neighborhood aggregation operator across different layers. Although doing so could exacerbate the vanishing and exploding gradient problems, gated graph neural networks alleviate this problem with the gating inherent in GRU units. The basic idea is that such units use gates in order to allow unimpeded gradient flow, which reduces the intensity of the vanishing and exploding gradient problems.

The first step is to set up a message from all the neighbors of a node. The message $\overline{m}_i^{(k)}$ from the neighbors of a node is the average of the all the embeddings of its neighbors in the previous time layer:

$$\overline{m}_i^{(k)} = \frac{\sum_{j \in A(i)} \overline{h}_j^{(k-1)}}{|A(i)|}$$

For simplicity, we assume that the hidden units in each time layer are p-dimensional. Then, the GRU update for each time layer is defined with the help of $2p \times 2p$ matrix U and $p \times 2p$ matrix V as follows:

$$(\forall i, k > 1): \quad \overline{h}_i^{(k)} = \text{GRU}(\overline{m}_i^{(k)}, \overline{h}_i^{(k-1)}, U, V)$$

Note that this update is based on the syntax of the GRU unit introduced in Equation 8.10 of Chapter 8. Therefore, the order and semantics of the parameters in the above equation is the same as that in Equation 8.10. It is noteworthy that this update is extremely parsimonious because the parameter matrices U and V are shared across all[3] hidden layers, and therefore the approach is quite parsimonious in terms of the number of parameters. Because of the use of gated units, one can construct very deep networks (in the range of 10 to 20 layers) without running into challenges such as the vanishing and exploding gradient problems. Such depth can bring great richness to the learning, as most real-world graphs do not have diameters greater than 7 to 10; therefore, significant amount of structural information of the graph can be learned with the use of 10 or more layers.

10.3.9 Comparison with Image Convolutional Networks

A convolutional neural network for image data can be considered a special type of graph neural network in which each pixel is the analog to a graph node; therefore, the "nodes" of an image are arranged in the form of a grid structure, and neighboring nodes are defined as those nodes that are convolved together within the spatial footprint of a filter. Just as a filter is shared across the entire image, the transformation matrices in graph neural networks are shared across all nodes. Just as deeper layers create embeddings with more global characteristics (and larger spatial footprint) of the image, deeper layers in graph neural networks capture features based on a larger neighborhood (and larger topological footprint) around a graph node. In fact, each convolutional layer roughly doubles the radius

[3]The only exception is at $k = 1$, where different matrices need to be used. This is because of the different size of the inputs as compared to the hidden layer dimensionality. Therefore, the sizes of the matrices would also need to be adjusted appropriately.

around a node used for creating the node features. The neighborhood region of the original graph captured by a particular node is referred to as its *receptive field*.

However, there are some key differences as well. The filter in an image convolutional network has parameters that are specific to the relative spatial position of a pixel within the region covered by the filter. Therefore, the size of a filter depends on the size of the spatial footprint it is convolved over. On the other hand, there is no ordering among the nodes in a graph and the neighborhood sizes of different nodes vary widely because of differences in node degrees. As a result, the parameters in a graph convolutional network do not have direct mapping to specific nodes in a neighborhood; rather the features of all the nodes in a neighborhood are added together (after normalization), and then multiplied with the parameter matrix. Therefore, the size of the parameter matrix is independent of the size of the local neighborhood being considered (unlike an image filter). Adding the representation of all neighborhood nodes before multiplying with parameter matrices leads to qualitatively different results, because it often leads to uncontrolled smoothing of the representations over a larger number of layers [300]. This is one of the reasons that graph convolutional networks tend to use a much smaller number of layers as compared to image convolutional networks [297]. Graph convolutional networks also have a harder time at reconstructing feature representations with autoencoder-like architectures, as smoothing can lead to irreversible loss of information. Emphasizing and deemphasizing different nodes (as image convolutional filters do with pixels) is critical for improving representation quality. This is often achieved with the use of *graph attention networks* (cf. section 12.2.8 of Chapter 12). The variation in the degrees of different nodes has some other consequences — it can often lead to less inputs for low-degree nodes, which often have poorly engineered representations and low classification accuracy [504].

Another important point is that graph neural networks generate embeddings at the *node* level rather than at the *graph* level, whereas most image convolutional network do so at the *image* level rather than the *pixel* level. In an image convolutional neural network, the analog of node embedding is pixel embedding. Although some convolutional neural networks (for images) are designed at the pixel level for specific applications, this is not the dominant use case. In image data, convolutional neural networks generate embeddings at the image level using a variety of methods for reducing the spatial footprint with operations like pooling. Operations like pooling enable hierarchical feature engineering. On the other hand, graph convolutional neural networks are inherently flat and are not well suited for hierarchical feature engineering. In order to reduce the size of the graph for representational purposes, a pooling operation is needed. In section 10.5, the design of pooling operations for graphs (and the corresponding generation of graph-level embeddings) will be discussed.

10.4 Backpropagation in Graph Neural Networks

In this section, we will discuss the backpropagation in graph neural networks. We will focus primarily on the neighborhood-based transform as an example, although the approach can be easily generalized to the graph convolutional transform. It is assumed that the graph neural network has T layers (not including the input layer), and each layer contains the neighborhood aggregation followed by the elementwise activation function. Therefore, the values of the hidden representations of the various layers (including the input layer) are $\overline{h}_i^{(0)}, \overline{h}_i^{(1)} \ldots \overline{h}_i^{(T)}$ for the ith node. These values represent *post-activation* values after the ac-

tivation function has already been applied. For simplicity, we will decouple the pre-activation and post-activation values, so that we can use the decoupled version of backpropagation in section 2.5.3 of Chapter 2. Therefore, the corresponding vectors of pre-activation values are denoted by $\overline{u}_i^{(0)}, \overline{u}_i^{(1)} \ldots \overline{u}_i^{(T)}$. One can then, decouple the simplest neighborhood-based transformations in Equation 10.3 to write it in terms of pre-activation and post-activation values:

$$(\forall i, k): \quad \overline{u}_i^{(k)} = W^{(k)} \sum_{j \in A(i)} \frac{\overline{h}_j^{(k-1)}}{|A(i)|} + B^{(k)} \overline{h}_i^{(k-1)} \qquad \text{[Pre-activation]}$$

$$\overline{h}_i^{(k)} = \Phi(\overline{u}_i^k) \qquad \text{[Postactivation]}$$

Let L is be the loss function of the neural network, and let $\overline{g}_i^{(k)}$ and $\overline{z}_i^{(k)}$ be the vector of gradients of the loss with respect to the pre-activation and the post-activation variables, respectively. Therefore, we have the following in matrix calculus notation:

$$(\forall i, k): \quad \overline{g}_i^{(k)} = \frac{\partial L}{\partial \overline{u}_i^{(k)}} \qquad \text{[Pre-activation derivative]}$$

$$\overline{z}_i^{(k)} = \frac{\partial L}{\partial \overline{h}_i^{(k)}} \qquad \text{[Postactivation derivative]}$$

Then, by using the rules of vector derivatives discussed in sections 2.5.1 and 2.5.3, the following backpropagation recurrences become evident:

$$(\forall i, k): \quad \overline{g}_i^{(k)} = \Phi'(\overline{u}_i^{(k)}) \odot \overline{z}_i^{(k)}$$

$$\overline{z}_i^{(k-1)} = \left[W^{(k)}\right]^T \sum_{j \in A(i)} \frac{\overline{g}_j^{(k)}}{|A(i)|} + \left[B^{(k)}\right]^T \overline{g}_i^{(k)}$$

Note that the backpropagation recurrences are very similar to forward propagation recurrences, except that the element-wise derivatives of the activation functions are used, and the weight matrices are transposed. This is similar to the decoupled view discussed in section 2.5.3.

So far, we have only discussed how to calculate derivatives with respect to activations in each layer. These derivatives with respect to the activations in each layer need to be converted into derivatives with respect to the weight matrices in each layer. Here, we only show derivative computation with respect to $W^{(k)}$ because the steps for computing the derivative with respect to $B^{(k)}$ are similar (see Exercise 6). The main issue to keep in mind is that the weight matrices are shared across the nodes in a given layer. Therefore, we first "pretend" that the weight matrices are not shared, and compute the derivative with respect to the ith node-specific copy $W_i^{(k)}$ of the weight matrix $W^{(k)}$. The derivative with respect to the ith node-specific copy is denoted by $M_i^{(k)}$, and it is of the same size as $W_i^{(k)}$. After computing the derivative with respect to the ith copy, all the derivatives with respect to the various copies are added (see section 2.6.6 of Chapter 2) to obtain the derivative matrix $M^{(k)}$ with respect to the weight matrix $W^{(k)}$ of the kth layer:

$$M^{(k)} = \sum_i M_i^{(k)}$$

How is the derivative matrix $M_i^{(k)}$ of the loss with respect to the ith node-specific copy $W_i^{(k)}$ of the weight matrix $W^{(k)}$ computed? This can be done easily using the pre-activation gradients in conjunction with the forward propagation equations:

$$M_i^{(k)} = \overline{g}_i^{(k)} \sum_{j \in A(i)} \frac{\left[\overline{h}_j^{(k-1)}\right]^T}{|A(i)|}$$

One can combine the above two equations to get a single expression for the derivative with respect to the weight matrix. Note that the right-hand side is in the form of an outer-product between two vectors, which creates a matrix. This equation is exactly analogous to Equation 2.23 of Chapter 2 used in conventional neural networks. A similar approach can also be used to compute the loss derivative with respect to the matrix $B^{(k)}$ regulating propagation over self connections, and we leave this problem as an exercise (see Exercise 6). We also leave the backpropagation procedure with respect to the graph convolution operator as an exercise (see Exercise 7).

10.5 Beyond Nodes: Generating Graph-Level Models

The similarity of graph convolutional networks to image convolutional networks leads to the question as to whether one can construct neural networks to process full graphs rather than simply embedding nodes. This ability is critical in applications such as drug discovery, where the properties of the drug are related to full graph representation of a chemical compound rather than individual nodes. In order to create such models, it is critical to be able to design pooling operations for graphs that are analogous to those in image data. This is because pooling operations help in creating a hierarchical feature engineering method, so that all graphs can be reduced to a fixed-length multidimensional representation. This fixed-length representation can then be hooked up with a conventional neural network for a task like classification. Note that both the conventional neural network and the graph neural network can be trained in an end-to-end manner with backpropagation.

There are several additional challenges associated with achieving the goal of pooling with nodes, when comparing with pooling pixels in images. A key point is that image pixels are clearly clustered *up front* based on spatial locality, whereas the notion of locality in graphs is somewhat more challenging and varies from one graph to another. One cannot simply choose a "spatial" region of nodes in a graph for coarsening, especially since one must use the same pooling model for a different input graph (with a possibly different number of nodes and no clear node-wise correspondence). Therefore, some notion of clustering is required, which is seamlessly embedded into the neural network model. Unlike pooling in images, one cannot make hard decisions *up front* as to which nodes are included in a particular pool (and this necessitates the integration of pooling decisions into the model itself). It is noteworthy that clustering is inherently a discrete optimization problem, and the tight integration of pool construction with model design causes problems for the differentiability required for neural network learning. This suggests that it makes sense to work with *soft* clusterings of nodes, in which nodes are defined with pre-defined probabilities. Soft clusterings can usually be modeled by continuous optimization models. Another problem is that the model constructed by the neural network *must generalize across multiple graphs* rather than a single graph. Therefore, any model for clustering the nodes (for pooling) must generalize

to multiple graphs, and it cannot represent the model for clustering of just a single graph. This is generally achievable when all the graphs belong to a particular domain with some typical characteristics.

In this section, we will discuss the architecture of a neural network for full graphs, and the pooling layer in it is referred to as *DiffPool*, which is short for Differentiable Pooling [568]. The architecture alternately arranges graph convolutional network modules and pooling layers in order to successively reduce the size of the graph. The convolutional module might itself require multiple iterations of message passing, and the specific details of these convolutional modules are abstracted out from the discussion.

It is assumed that the input to the architecture is the $n_0 \times n_0$ adjacency matrix A_0 and $n_0 \times d$ feature matrix H_0; the latter can also be viewed as the hidden representation of the zeroth layer (even though it is an input $n \times d$ matrix). The number of nodes in the coarsened network after k layers is denoted by n_k. Therefore, we will always have $n_{k+1} < n_k$ as a result of the pooling. Correspondingly, we will refer to the hidden representations of the kth layer just after pooling by the $n_k \times n_k$ (coarsened) adjacency matrix A_k and the $n_k \times p_k$ feature matrix H_k, respectively. One can assume that each A_k is a weighted adjacency matrix. Furthermore, since graph convolutional modules and pooling layers alternate, the feature representation matrix after the kth graph convolutional module is the $n_k \times p_k$ feature matrix F_k. The feature representation after the kth pooling module is H_k. The main difference between F_k and H_k is that the former is the output of a convolutional module, whereas the latter is the output of a pooling layer. These two layers alternate with one another. Therefore, the sequence of feature representations of the graph (in matrix form) is as follows:

$$H_0, F_1, H_1, F_2, H_2 \ldots F_k, H_k \ldots F_T, H_T$$

The number of rows in these matrices reduce in size with increasing subscript index, as the nodes get pooled into 'supernodes'. Technically, the adjacency matrix and the pooled cluster assignment matrix are also generated along the way. Therefore, we have the following:

$$[A_0, H_0], [S_1, F_1], [A_1, H_1], [S_2, F_2], [A_2, H_2] \ldots [S_k, F_k], [A_k, H_k], \ldots, [S_T, F_T], [A_T, H_T]$$

The final layer in the graph neural network (just before the conventional neural network) is always a pooling layer, since the above sequence ends with $[A_T, H_T]$. The matrix A_T is always an adjacency matrix with one node, and the matrix H_T is always a $1 \times d$ matrix with a fixed length representation of d. Therefore, there will be a total of T convolution-centric transforms and T pooling operations. An example of the architecture at $T = 3$ is shown[4] in Figure 10.3. In the following, we will discuss how these matrices are sequentially generated via various matrix operations and modules.

A key requirement for performing pooling for the graph in the kth layer is an $n_k \times n_{k+1}$ cluster assignment matrix S_{k+1} that maps n_k nodes to n_{k+1} clusters (which become the nodes of the next layer after pooling of cluster nodes). The ith row of this matrix contains the probability with which the ith node in the original/coarsened graph A_k is assigned to each cluster. Therefore, each row must sum to 1. Note that this is a soft clustering assignment in which each of the n_k nodes has a non-zero probability of belonging to each cluster. Given the adjacency matrix A_k and the feature matrix H_k, a new $n_k \times p_{k+1}$ feature matrix F_{k+1} and the $n_k \times n_{k+1}$ assignment matrix S_{k+1} are learned via two separate graph convolutional networks with distinct parameters. We refer to these convolutional networks as the feature transformation module GCN_k^F and the pooling module GCN_k^P, respectively,

[4] An earlier version of this figure appeared in [313] in the context of a somewhat different algorithm.

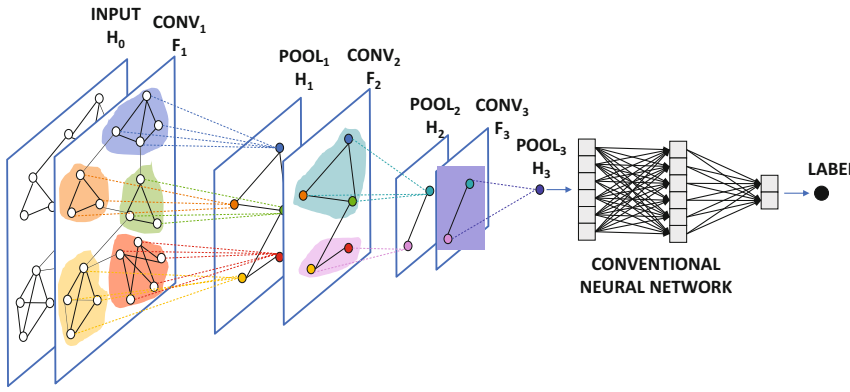


Figure 10.3: Alternating pooling and convolution in *DiffPool*. ©Yao Ma and Jiliang Tang. Used with permission.

for the kth layer. In other words, we have the following:

$$[F_{k+1}]_{n_k \times p_{k+1}} = \text{GCN}_k^F(A_k, H_k) \qquad \text{[New feature matrix]}$$
$$[S_{k+1}]_{n_k \times n_{k+1}} = \text{Softmax}\left[\text{GCN}_k^P(A_k, H_k)\right] \qquad \text{[Cluster assignment matrix]}$$

The softmax function is applied row-wise to the $n_k \times n_{k+1}$ output of the graph convolutional network module, so that each row of cluster assignment probabilities sums to 1. Armed with this cluster assignment matrix, one can then generate the pooled adjacency matrix A_{k+1} and the feature matrix H_{k+1} for this smaller graph are generated. The pooled adjacency matrix is constructed as follows:

$$[A_{k+1}]_{n_{k+1} \times n_{k+1}} = S_{k+1}^T A_k S_{k+1}$$

By using this approach to pooling, one is creating an $n_{k+1} \times n_{k+1}$ adjacency matrix A_{k+1} of supernodes, in which one is aggregating the weights of edges between all pairs of nodes in the same cluster pair in order to create new weights. Self-loops are created because of nodes in the same cluster. Furthermore, the aggregation is done in a soft way with probability weights present in S_k. Furthermore, the new $n_{k+1} \times p_{k+1}$ feature matrix H_{k+1} of this pooled adjacency matrix A_{k+1} is generated as follows:

$$[H_{k+1}]_{n_{k+1} \times p_{k+1}} = S_{k+1}^T F_{k+1}$$

This transformation simply aggregates the features in each cluster, while weighting the features of each node with its cluster assignment probability. Note that the *DiffPool* operation is completely differentiable because of the use of soft cluster assignment probabilities. In order to ensure that each row of matrix S_k produces probabilities that are large for a small number of clusters, the cross-entropy loss $E(S_k)$ of the node assignment for each matrix S_k is aggregated over all k and added to the loss function of the application at hand:

$$E(S_k) = -\sum_i \sum_j s_{ij}^{(k)} \ln(s_{ij}^{(k)})$$

Here, $s_{ij}^{(k)}$ is the (i, j)th entry of S_k. This loss takes on the least value of 0, when the soft clustering assignment is close to a one-hot encoding in each row.

The cluster assignment matrix S_T that occurs just before the final pooling operation is special, in the sense that is not learned but it is hard-coded to an $n_k \times 1$ matrix of 1s. This ensures that the entire graph is reduced to one node with a 1×1 "adjacency matrix" with a self-loop, and the feature vectors of all nodes in the penultimate layer are aggregated to create a fixed-length representation with d features. Therefore, the matrix H_T is of size $1 \times d$, and it contains the d-dimensional multidimensional representation of the entire graph. This representation can be input to a conventional neural network for application-specific tasks like classification or regression. Note that this architecture is quite complex, since it contains multiple graph convolutional neural network *modules*, pooling layers, as well as a conventional network. All these components are trained with backpropagation in an end-to-end manner. The approach can be used directly for classification of test graphs, or even to extract embeddings of individual graphs from the final pooled node.

Unsupervised Autoencoder

Unsupervised embeddings can also be generated with the use of an autoencoder architecture that is designed with the use of *unpooling* operations in addition to pooling operations. Technically, the unpooling operation can be considered a lossy inversion of the pooling operation. In order to make the architecture appropriate for supervised learning, one can design a symmetric architecture with corresponding embeddings F'_k and H'_k in the decoder, in which features F'_{k+1} unpool to H'_{k+1}. This architecture is shown in Figure 10.4. An important issue is that unpooling operations need to be mirrored from the pooling variables in the encoder so that pooling and unpooling correspond to one another. This matrix can be obtained by copying S_k from the encoder to the decoder, normalizing it columnwise, and then transposing it to create a wide matrix S'_k. This copying is illustrated by dashed lines in Figure 10.4. The S'_k matrix can be used to unpool (disaggregate clusters) with the following relationships:

$$F'_{k+1} = [S'_{k+1}]^T H'_{k+1}$$
$$A'_k = [S'_{k+1}]^T A'_{k+1} [S'_{k+1}]$$

Note that S'_{k+1} is the left-inverse of S_{k+1} in the ideal case where each row of S_{k+1} is one-hot encoded, and $S_{k+1}[S'_{k+1}]^T$ has a special structure, referred to as a *projection matrix* (in this ideal case). What this means is that applying only the pooling and unpooling operations to an adjacency matrix without convolution leads to a lossy projection of the graph adjacency matrix to one with rank equal to the number of clusters (in the ideal case of single-cluster membership of each node). The primary loss function is the squared error between the feature variables of the corresponding nodes in the original graph and the reconstructed graph. It is also possible to allow the innermost layer to have a number of nodes that is a specific fraction of that in the input graph (rather than reducing to a single node). This can potentially allow better reconstruction when the graphs are too large and reducing to one node loses too much information. Although the resulting (embedding) matrix in the innermost layer will not be of fixed size over different graphs, the rows of this matrix can be averaged offline to a fixed length representation for various application-specific tasks like classification.

A different approach, referred to as *Graph U-Net* [136], uses node downsampling and upsampling for the pooling and unpooling operations (instead of cluster aggregation and disaggregation). Sampling requires fewer parameters than clustering to capture the pooling

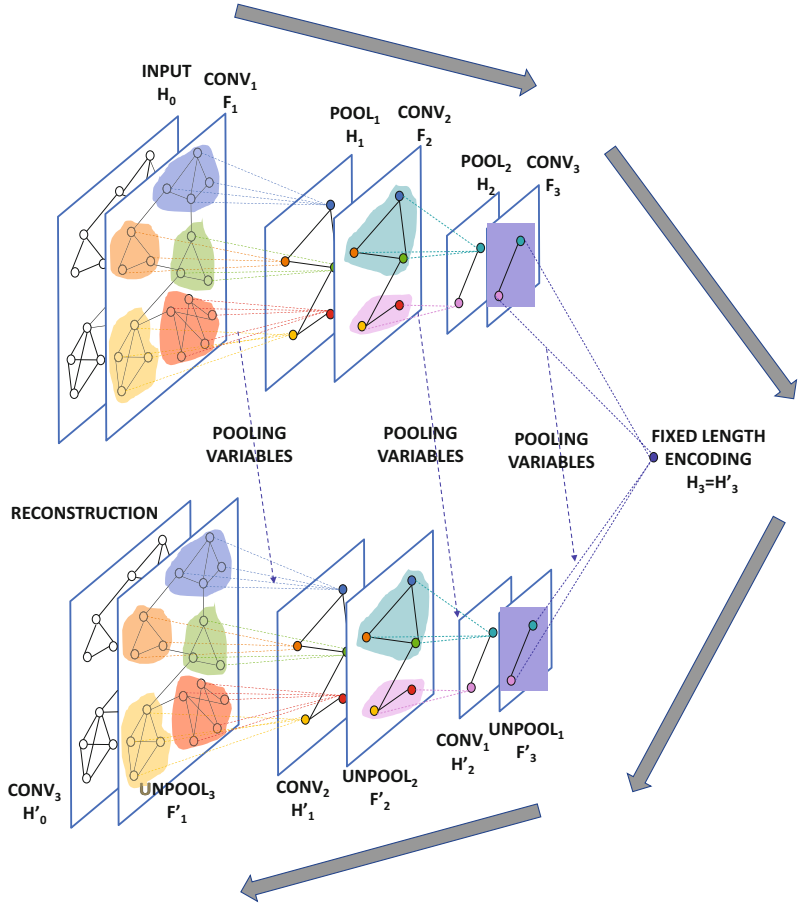


Figure 10.4: Unsupervised graph autoencoder. ©Yao Ma, Jiliang Tang, and Charu Aggarwal. Used with permission.

operation. Specifically, a trainable (unit) projection vector $\overline{s}_k$ of size p_k is used in lieu of the $n_k \times n_{k+1}$ matrix S_k in order to create a list of the indices of largest n_{k+1} entries of the n_k-dimensional column vector $F_{k+1}\overline{s}_{k+1}$ during pooling, and the corresponding rows with these indices are included in H_{k+1} in order to create the $n_{k+1} \times p_k$ matrix H_{k+1}. Furthermore, each row of the pooled matrix H_{k+1} is scaled with the node's corresponding value in the n_k-dimensional vector $[\text{sigmoid}(F_{k+1}\overline{s}_{k+1})]$. This operation also makes the pooling process differentiable. During unpooling, all rows of the smaller matrix H'_{k+1} are placed back in their original row-index positions (from which they were extracted during pooling) to generate F'_{k+1} (of the same size as F_{k+1}); the remaining rows of the matrix F'_{k+1} are padded with zero vectors. Note that the placement information corresponding to the list of pooled row indices is needed to do this, and it is therefore transmitted from each pooling layer to the corresponding unpooling layer using the dotted connections of Figure 10.4. Sampling-centric pooling might lead to breakages in graph connectivity, and therefore the "sampled" adjacency matrix A'_{k+1} for the next layer is generated from pooling by selecting the sampled rows and columns of the (denser) second-order walk matrix A_k^2 instead of A_k. Skip connections are added between the corresponding convolved representations in the

encoder and decoder for better accuracy. The convolved representations of the encoder can be concatenated or added to those in the decoder. The Graph U-Net approach does not have a final layer aggregating the different nodes into a single node, and therefore the reduced representation might have variable length over different input graphs.

10.6 Applications of Graph Neural Networks

The most direct use of graph neural networks is for node classification, which has already been discussed earlier in this chapter. There are also numerous uses of the embeddings that are constructed from graph neural networks. This section will discuss some of these applications. The embeddings in different hidden layers provide embeddings with different properties. The nodes that occur in earlier layers typically learn very local (and simple) features, whereas the nodes in later layers tend to learn more complex (and global) features. It is possible to concatenate the features in the last two or three layers in order to create a single embedding. Even though an embedding is typically not needed for applications like classification, other related applications like clustering and link prediction greatly benefit from embeddings.

One of the challenges in machine learning from graph data is that the representations of graph data contain diverse characteristics such as structure and application-specific attributes, which are not quite simple to use with most off-the-shelf machine learning algorithms. While there are dedicated algorithms for applying machine learning algorithms to graphs, the presence of application-specific attributes along with structural characteristics greatly reduces the number of algorithms that are directly usable. On the other hand, most known machine learning algorithms have been developed for multidimensional data. Therefore, these embeddings naturally generalize the use of many off-the-shelf machine learning algorithms (for multidimensional data) to graph data.

The embeddings that are constructed from graph neural networks can be leveraged in a wide variety of graph-centric applications such as clustering, link prediction, and outlier detection. In other words, the embedding may be used in a wide variety of tasks. This general principle is illustrated in Figure 10.5. The applications that are supported by graph convolutional networks are discussed below.

Classification

Of all the tasks supported by graph convolutional networks, this is perhaps the most common task. In most cases, the embedding does not need to be extracted from the hidden

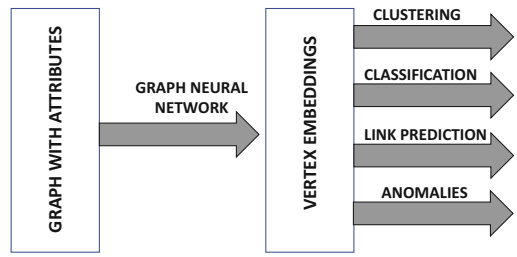


Figure 10.5: The wide spectrum use of graph embeddings in machine learning

layers in order to classify nodes. For most such settings, the graph convolutional network is trained on a labeled network, and it directly predicts the labels of unlabeled nodes with the use of a softmax layer. A discussion of how the output layer and the loss function can be structured for the classification problem is provided in earlier sections. The approach can also be used for classification of full graphs (rather than nodes) with the use of the architecture in Figure 10.3.

Clustering

In clustering, the goal is to create groups of nodes that are similar to one another either in terms of the local structure, types of neighbors, and the properties of nodes. While there are many algorithms that can directly discover clusters on graph data, the presence of attribute-specific data (e.g., social network profile) along with the graph structure (e.g., adjacency matrix) greatly reduces the number of applications one can directly use for clustering purposes. On the other hand, once a multidimensional embedding has been created from the nodes, one can use any existing clustering algorithm (e.g., k-means) in order to create groups of similar nodes from the graphs.

Anomaly Detection

The anomaly detection is closely related to clustering, wherein we try to find nodes that are fundamentally different from other nodes in the graph. The underlying causalities of anomalies in a graph may be very complex. For example, a node may be an anomaly because it has very high degree, or a node may be an anomaly because it is connected to nodes from many different communities. For example, in a publication network, a researcher who is highly connected to the communities in the math/engineering disciplines as well as the liberal arts disciplines is highly unusual and may be considered an anomaly. Obviously, such anomalies cannot be easily discovered by using only the network structure, unless one already has some domain knowledge of what the "natural" structure of networks should be. On the other hand, the multidimensional embeddings of graph neural networks already encode the key structural characteristics of graphs within the attributes of the embedding. Therefore, simple multidimensional outlier detection algorithm (e.g., reporting the k-nearest neighbor distance of a node embedding from other embeddings as the outlier score) can discover such anomalous characteristics in a straightforward way.

Link Prediction and Recommendations

In the link prediction problem the goal is to find pairs of nodes that are not currently connected, but have a propensity to become connected in the future. For example, in a social network, two actors who are connected to the same set of people are more likely to become connected in the future. Similarly, if their profile indicates that they have the same alma mater, they are also likely to become connected. Therefore, similar nodes tend to become connected based on the principle of *homophily* (i.e., to like the similar), although it is generally difficult to discover the similar in a network without some domain knowledge of the network at hand. The multidimensional embeddings generated by graph neural networks can be very helpful in discovering such similar actors, because the embeddings already incorporate the most typical characteristics of connectivity between actors. Therefore, a simple k-nearest neighbor algorithm on the multidimensional embedding can often provide high-quality link predictions.

The recommendation problem is closely related to link prediction because every recommendation problem can be formulated as a link prediction problem. Consider a set of users and items, where we have information about the buying behavior of the users for various items. Our goal is to recommend items to users that they have not bought before. Furthermore, various types of content attributes may be associated with users and items, such as the user profile or the item profile.

All recommendation problems can be formulated as link prediction problems on *bipartite graphs*. In bipartite graphs, nodes can be partitions into two sets such that edges exist between nodes belonging to the sets, but not within the nodes of a particular set. In the case of the recommendation problem, the nodes correspond to users and items, respectively. A node is created for each user, and a node is created for each item. A user node is connected to an item node, if and only if the user had bought an item earlier. Then, the recommendation problem of items for users can be simply formulated as that of finding missing links between users and items. As discussed above, this problem can be solved by finding the embeddngs of both users and items. For each user, one can find embeddings of similar items by using a nearest neighbor algorithm. These items can be recommended to the user.

It is noteworthy that this framework can even be extended to social network recommendations, where are social connections among the users. In this case, we have two types of links, corresponding to user-item (activity) connections and user-user (social) connections. The problem boils down to finding the embeddings of nodes in a relational network (i.e., heterogeneous information network) setting, which has been discussed in this chapter. One can use the embedding to recommend either items to users (by finding the nearest-neighbor items) or to recommend users to other like-minded users (by finding nearest-neighbor users).

10.7 Summary

This chapter discusses the design of different types of graph neural networks. The simplest type of neural network for embedding nodes is a generalization of the *word2vec* framework. However, in order to learn rich characteristics of graphs, it is important to create deep neural networks that use the structure of the graph in creating the neural network. Therefore, the broad inspiration of graph neural networks is obtained from recurrent networks where parameter matrices are shared across different layers and data items (sequence elements). In simple versions of graph neural networks, the parameter matrices are shared across different data items (nodes) but not across different time layers. However, such an approach can be only two or three levels deep. In order to make graph neural networks deep, many ideas from gated recurrent units have been adapted to graph neural networks. Methods have also been designed for graph pooling and unpooling, which are useful for embedding full graphs. The embeddings generated from graph neural networks have numerous applications, such as clustering, classification, anomaly detection, link prediction, and recommendations.

10.8 Bibliographic Notes and Software Resources

Overviews on graph neural networks may be found in tutorials by Hamilton *et al.* [177] and Karypis *et al* [587]. A video version of the latter tutorial is available [587], and it covers hands-on training of the use of a graph representation learning library. Excellent surveys on the topic may be found in [175] and [558].

The basic idea of shallow graph embedding has been covered in algorithms like *word2vec* [171] and *DeepWalk* [385]. A key point about the use of such shallow networks is that it becomes more critical to use higher order adjacency matrices, such as using he kth power A^k of the adjacency matrix rather than the adjacency matrix directly.

Graph neural networks do not perform such feature engineering, but they instead work with the original adjacency matrix. The basic idea of neighborhood aggregation was based on early work on the graph neural network model by Scarselli *et al.* [444]. The graph convolutional network model was first presented in [253]. The modeling of relational data with graph convolutional networks is discussed in [314, 448]. Other deep architectures for heterogeneous network embedding are discussed in [63]. The GraphSage model is presented in [176]. Various types of sampling methods have also been proposed to speed up graph convolutional networks and also reduce variance [65, 66]. The problems of over-smoothing in graph neural networks and corresponding issues with deep graph neural networks are discussed in [297, 300]. The effects of degree-related biases are discussed in [504]. The development of deep networks with the use of gating is discussed in [302]. There is some debate on whether the complexity of deep graph networks is really needed, and whether one can achieve the same results as shallow networks with feature engineering (e.g., using multi-hop features or random walks) [553]. The *DiffPool* approach is adapted from [568], and is generally designed for supervised learning. The use of eigenpooling for improved supervised learning is available in [313]. Unsupervised learning also requires unpooling operations, which are discussed in [136]. Graph attention networks are discussed in [525], and this class of methods is introduced in section 12.2.8 of Chapter 12.

Numerous applications of graph neural networks are discussed in the WWW tutorial of [177]. The use of graph neural networks for Web-scale and social recommender systems is discussed in [119, 567]. The use of deep network embedding for anomaly detection in discussed in [572, 573]. Applications of full-graph models for drug discovery are discussed in [145].

Software Resources and Data Sets

Numerous network data sets are available from the SNAP Stanford Data set repository [667]. The code for GraphSAGE is also downloadable from [668]. The code based on the original implementation of graph convolutional networks is available from [253]. The author's original implementation of *DiffPool* is available at [568]. A list of TensorFlow implementations of various types of graph neural networks are available in [669]. The deep graph library also contains several implementations for representation learning on graphs [670]. Another popular library containing various graph neural networks is the Geometric Deep Learning Extension Library for PyTorch [671].

10.9 Exercises

1. Consider a graph in which you have application-specific node attributes available, which are multidimensional (e.g., user age and gender in a social network). Discuss how you can modify the architecture of Figure 10.1 in order to create embeddings that account for these additional node attributes.

2. Consider a relational graph in which the edges are of different types. For example, in a database containing movies, actors, and directors, one might have links such as

acted in and *directed in*. Discuss how you can change the outputs of Figure 10.1 in order to account for these types of heterogeneous edges.

3. A knowledge graph might contain both attributes associated with nodes as well as edges of different types. This is precisely the scenario that corresponds to combining the conditions of Exercises 1 and 2. Discuss how you would modify the architecture of Figure 10.1 in order to accommodate links of different types as well as node attributes.

4. Suppose that you have a graph in which every node corresponds to a protein, and edges correspond to degree-of-similarity relationships between proteins. Each node contains a sentence (or caption) describing its function. No other attributes are associated with nodes. Discuss how you can combine a graph neural network with a recurrent neural network in order to generate an embedding of each node that accounts for its sentence description and relationships with other proteins. Nw suppose that a new node is added to the network along with edges to nodes that the new protein is most similar to. How would you use the aforementioned neural network structure in order generate a caption for this new node?

5. Suppose that you different networks (e.g., Facebook and Twitter) that share some nodes in common (that are known). Discuss why embedding each node in the network in a shared space with the use of a graph convolutional network is straightforward based on the ideas discussed in this chapter.

6. Section 10.4 discusses the backpropagation algorithm for neighborhood-based aggregation, and also shows how to compute the loss derivative with respect to the weight matrices $W^{(k)}$ defining propagations from neighbor connections. However, the loss derivative with respect to the matrix $B^{(k)}$ regulating propagations over self connections is not discussed. Propose the equation describing the loss derivative with respect to $B^{(k)}$ in terms of the gradients with respect to the layers and layer activations.

7. Generalize the algorithm discussed in section 10.4 for the simple neighborhood-based operator to the graph convolution operator.

8. Discuss how the graph convolution operator of Equation 10.5 can be generalized with a single hyper-parameter in order to allow greater control on the importance of self connections in the convolution operation.

9. **Open-ended:** Consider a graph in which nodes have types but edges do not have types (or are not specified in the data). How would you modify the model of section 10.3.6 to work for this case? How would you handle cases in which different node types have completely different attributes?

10. **Open ended:** Consider a collection of objects for which you do not know the representation. You are only given a pair-wise similarity function between them. Discuss how you would modify the model of section 10.2 to learn a representation for each node.

11. Discuss the conceptual similarity between GraphSAGE and the notion of skip connections in ResNets.

12. **Open-ended:** Consider a recommendation application in which you have users with user attributes, items with item attributes, and a sparse matrix of ratings between users and items. The user attributes and item attributes may be quite different in semantic interpretation. Discuss how you can construct a graph convolutional network that leverages all this information. Describe the structure of the graph convolutional network as well as that of the output layer.

Chapter 11

Deep Reinforcement Learning

"The reward of suffering is experience."–Harry S. Truman

11.1 Introduction

Human beings do not learn from a concrete notion of training data. Learning in humans is a continuous experience-driven process in which decisions are made, and the reward/punishment received from the *environment* are used to guide the learning process for future decisions. In other words, learning in intelligent beings is by reward-guided *trial and error*. Furthermore, much of human intelligence and instinct is encoded in genetics, which has evolved over millions of years with another environment-driven process, referred to as *evolution*. Therefore, almost all of biological intelligence, as we know it, originates in one form or other through an interactive process of trial and error with the environment. In his interesting book on artificial intelligence [473], Herbert Simon proposed the *ant hypothesis*:

> "Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves."

Human beings are considered simple because they are one-dimensional, selfish, and reward-driven entities (when viewed as a whole), and all of biological intelligence is therefore attributable to this simple fact. Since the goal of artificial intelligence is to simulate biological intelligence, it is therefore natural to draw inspirations from the successes of biological greed in simplifying the design of highly complex learning algorithms.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_11

A reward-driven trial-and-error process, in which a system learns to interact with a complex environment to achieve rewarding outcomes, is referred to in machine learning parlance as *reinforcement learning*. In reinforcement learning, the process of trial and error is driven by the need to maximize the expected rewards over time. Reinforcement learning can be a gateway to the quest for creating truly intelligent *agents* such as game-playing algorithms, self-driving cars, and even intelligent robots that interact with the environment. Simply speaking, it is a gateway to general forms of artificial intelligence. We are not quite there yet. However, we have made huge strides in recent years with exciting results:

1. Deep learners have been trained to play video games by reinforcement learning. The input to the deep learner is the display of pixels from the current state of the game. The reinforcement learning algorithm predicts the actions based on the display and inputs them into the video game console. Initially, the computer algorithm makes many mistakes, which are reflected in the virtual rewards given by the console. As the learner gains experience from its mistakes, it makes better decisions. This is exactly how humans learn to play video games. Video games are excellent test beds for reinforcement learning algorithms, because they can be viewed as highly simplified representations of the choices one has to make in various decision-centric settings. Simply speaking, video games represent toy microcosms of real life.

2. DeepMind has trained a deep reinforcement learning algorithm *AlphaZero* [465, 467] to play the game of chess, shogi, and *Go* by using the experience gained from computer self-play. *AlphaZero* has not only convincingly defeated human players, but has contributed to innovations in the style of human play. These innovations were a result of the reward-driven experience gained by *AlphaZero* by playing itself over time (just as humans learn innovations via practice).

3. In recent years, deep reinforcement learning has been harnessed in self-driving cars by using the feedback from various sensors around the car to make decisions [628]. Such cars now consistently make fewer errors than do human beings.

4. The quest for creating self-learning robots is a task in reinforcement learning [292, 305, 451]. In order to teach a robot to walk, we incentivize the robot to get from point A to point B as efficiently as possible using its available limbs and motors [451]. Through reward-guided trial and error, robots learn to roll, crawl, and eventually walk.

Reinforcement learning is appropriate for tasks *that are simple to evaluate but hard to specify*. For example, it is easy to evaluate a player's performance at the end of a complex game like chess, but it is hard to specify the precise action in every situation. As in biological organisms, reinforcement learning provides a path to the *simplification of learning complex behaviors* by only defining the reward and letting the algorithm learn reward-maximizing behaviors. The complexity of these behaviors is automatically inherited from that of the environment. This is the essence of Herbert Simon's ant hypothesis [473] at the beginning of this chapter. Reinforcement learning systems are inherently *end-to-end systems* in which a complex task is not broken up into smaller components, but viewed through the lens of a simple reward.

Chapter Organization

This chapter is organized as follows. The next section introduces multi-armed bandits, which is the simplest settings of reinforcement learning. The notion of states is introduced

in section 11.3. Monte Carlo sampling algorithms are discussed in section 11.4. Bootstrapping methods are introduced in section 11.5. Policy gradient methods are discussed in section 11.6. The use of Monte Carlo tree search strategies is discussed in 11.7. A number of case studies are discussed in section 11.8. The safety issues associated with deep reinforcement learning methods are discussed in section 11.9. A summary is given in section 11.10.

11.2 Stateless Algorithms: Multi-Armed Bandits

The simplest example of a reinforcement learning setting is the *multi-armed bandit problem*, which addresses the problem of a gambler choosing one of many slot machines in order to maximize his payoff. The gambler suspects that the (expected) rewards from the various slot machines are not the same, and therefore it makes sense to play the machine with the largest expected reward. Since the expected payoffs of the slot machines are not known in advance, the gambler has to *explore* different slot machines by playing them and also *exploit* the learned knowledge to maximize the reward. Multi-armed bandit algorithms provide carefully crafted strategies to optimize the trade-off between exploration and exploitation.

Although exploration of a particular slot machine might gain some additional knowledge about its payoff, it incurs the risk of the (potentially fruitless) cost of playing it. Trying the slot machines randomly is wasteful but helps in gaining experience. Trying the slot machines for a very small number of times and then always picking the best machine might lead to solutions that are poor in the long-term. How should one navigate this trade-off between exploration and exploitation? Note that every trial provides the same probabilistically distributed reward as previous trials for a given action, and therefore there is no notion of *state* in such a system (unlike chess where the action depends on the board state). This is the simplest case of reinforcement learning.

In the following, we will briefly describe some of the common strategies used in multi-armed bandit systems to regulate the trade-off between exploration and exploitation. All these methods are instructive because they provide the basic ideas and framework, which are used in generalized settings of reinforcement learning. In fact, some of these stateless algorithms are also used to define state-specific policies in general forms of reinforcement learning. Therefore, it is important to explore this simplified setting.

Naïve Algorithm

In this approach, the gambler plays each machine for a fixed number of trials in the exploration phase. Subsequently, the machine with the highest payoff is used forever in the exploitation phase. Although this approach might seem reasonable at first sight, it has a number of drawbacks. The first problem is that it is hard to determine the number of trials at which one can confidently predict whether a particular slot machine is better than another machine. The process of estimation of payoffs might take a long time, especially in cases where the payoff events are rare compared to non-payoff events. Using many exploratory trials will waste a significant amount of effort on suboptimal strategies. Furthermore, if the wrong strategy is selected in the end, the gambler will use the wrong slot machine forever. Therefore, the approach of fixing a particular strategy forever is unrealistic in real-world problems.

ϵ-Greedy Algorithm

The ϵ-greedy algorithm is designed to use the best strategy as soon as possible, without wasting a significant number of trials. The basic idea is to choose a random slot machine for a fraction ϵ of the trials. These exploratory trials are also chosen at random (with probability ϵ) from all trials, and are therefore fully interleaved with the exploitation trials. In the remaining $(1-\epsilon)$ fraction of the trials, the slot machine with the best average payoff so far is used. An important advantage of this approach is that one is guaranteed to not be trapped in the wrong strategy forever. Furthermore, since the exploitation stage starts early, one is often likely to use the best strategy a large fraction of the time.

The value of ϵ is an algorithm parameter. For example, in practical settings, one might set $\epsilon = 0.1$, although the best choice of ϵ will vary with the application at hand. It is often difficult to know the best value of ϵ to use in a particular setting. Nevertheless, the value of ϵ needs to be reasonably small in order to gain significant advantages from the exploitation portion of the approach. However, at small values of ϵ it might take a long time to identify the correct slot machine. A common approach is to use *annealing*, in which large values of ϵ are initially used, with the values declining with time.

Upper Bounding Methods

Even though the ϵ-greedy strategy is better than the naïve strategy in dynamic settings, it is still quite inefficient at learning the payoffs of new slot machines. In upper bounding strategies, the gambler does not use the mean payoff of a slot machine. Rather, the gambler takes a more optimistic view of slot machines that have not been tried sufficiently, and therefore uses a slot machine with the best *statistical upper bound* on the payoff. Therefore, one can consider the upper bound U_i of testing a slot machine i as the sum of expected reward Q_i and one-sided confidence interval length C_i:

$$U_i = Q_i + C_i \tag{11.1}$$

The value of C_i is like a bonus for increased uncertainty about that slot machine in the mind of the gambler. The value C_i is proportional to the standard deviation of the *mean* reward of the tries so far. According to the central limit theorem, this standard deviation is inversely proportional to the square-root of the number of times the slot machine i is tried (under the i.i.d. assumption). One can estimate the mean μ_i and standard deviation σ_i of the ith slot machine and then set C_i to be $K \cdot \sigma_i / \sqrt{n_i}$, where n_i is the number of times the ith slot machine has been tried. Here, K decides the level of confidence interval. Therefore, rarely tested slot machines will tend to have larger upper bounds (because of larger confidence intervals C_i) and will therefore be tried more frequently.

Unlike the ϵ-greedy algorithm, the trials are no longer divided into two categories of exploration and exploitation; the process of selecting the slot machine with the largest upper bound has the dual effect of encoding both the exploration and exploitation aspects within each trial. One can regulate the trade-off between exploration and exploitation by using a specific level of statistical confidence. The choice of $K = 3$ leads to a 99.99% confidence interval for the upper bound under the Gaussian assumption. In general, increasing K will give large bonuses C_i for uncertainty, thereby causing exploration to comprise a larger proportion of the plays compared to an algorithm with smaller values of K.

11.3 The Basic Framework of Reinforcement Learning

Bandit algorithms are stateless. In other words, the decision made at each time stamp has an identical environment, and the actions in the past only affect the knowledge of the agent (and not the environment). This is not the case in generic reinforcement learning settings like video games or self-driving cars, which have a notion of *state*. While playing a video game, the reward for moving the joystick in a certain way (action) depends on all the other actions done in the past, which are incorporated in the *state* of the pixels of the video game (environment). Similarly, in a self-driving car, the reward for violently swerving a car in normal traffic conditions would be drastically different from that in a situation indicating the danger of a collision. Therefore, the evaluation of actions needs to be *state-sensitive*. Some examples of rewards in reinforcement learning settings are as follows:

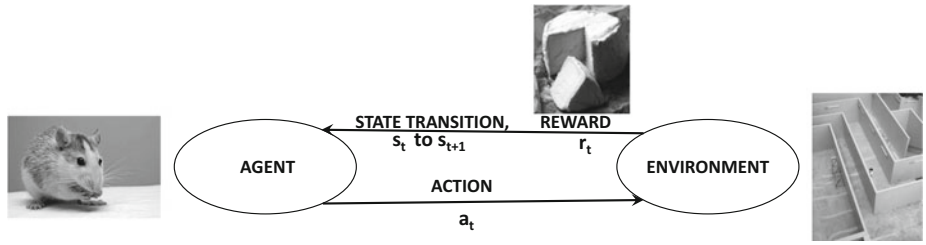
1. *Game of tic-tac-toe, chess, or Go:* The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward is $+1$, 0, or -1 (depending on win, draw, or loss), *which is received at the end of the game*. Note that rewards are often not received immediately after strategically astute actions.

2. *Robot locomotion:* The state corresponds to the current configuration of robot joints and its position. The actions correspond to the torques applied to robot joints. The reward at each time stamp is a function of whether the robot stays upright and the amount of forward movement from point A to point B.

3. *Self-driving car:* The states correspond to the sensor inputs from the car, and the actions correspond to the steering, acceleration, and braking choices. The reward is a hand-crafted function of car progress and safety.

In reinforcement learning, the virtual entity that interacts with the environment to perform actions and earn rewards is referred to as an *agent* (such as a video-game player). The actions change the state of the environment (such as moving the joystick changing the game state). The environment gives the agent rewards (such as virtual points). The consequence of an action is sometimes long lasting. For example, the agent might have cleverly positioned a cursor at a particularly convenient point a few moves back, and the current action may yield a high reward only because of that positioning. Therefore, not only does the current state-action pair need to be credited, but the past state-action pair (e.g., positioning the cursor) needs to be appropriately credited when a reward (e.g., video game point) is received. Furthermore, the reward for an action might not be deterministic (as in card games), which adds to complexity. *One of the primary goals of reinforcement learning is to identify the inherent values of actions in different states, irrespective of the timing and stochasticity of the reward.* The agent can then choose actions based on these values.

This general principle is drawn from how reinforcement learning works in biological organisms. Consider a mouse learning a path through a maze to earn a reward. The value of a particular action of the mouse (e.g., left turn) depends on where it is in the maze. When a reward is earned by reaching the goal, the synaptic weights in the mouse's brain adjust to reflect all past actions in various positions and not just the final step. This is exactly the approach used in deep reinforcement learning, where a neural network is used to predict values of actions from sensory inputs (e.g., pixels of video game), and the values of past actions in various states are indirectly updated depending on the received rewards (by updating the weights of the neural network). This relationship between the agent and the environment is shown in Figure 11.1.



1. AGENT (MOUSE) TAKES AN ACTION a_t (LEFT TURN IN MAZE) FROM STATE (POSITION) s_t
2. ENVIRONMENT GIVES MOUSE REWARD r_t (CHEESE/NO CHEESE)
3. THE STATE OF AGENT IS CHANGED TO s_{t+1}
4. MOUSE'S NEURONS UPDATE SYNAPTIC WEIGHTS BASED ON WHETHER ACTION EARNED CHEESE

OVERALL: AGENT LEARNS OVER TIME TO TAKE STATE-SENSITIVE ACTIONS THAT EARN REWARDS

Figure 11.1: The broad framework of reinforcement learning

The process of generating a sequence of states, actions, and rewards is referred to as a *Markov decision process*. The main property of a Markov decision process is that the state at any particular time stamp encodes all the information needed by the environment to make state transitions and assign rewards based on agent actions. Finite Markov decision processes (e.g., tic-tac-toe) terminate in a finite number of steps, which is referred to as an *episode*. Infinite Markov decision processes (e.g., continuously working robots) do not have finite length episodes and are referred to as *non-episodic* or *continuous*. A Markov decision process can be represented as a sequence of actions, states, and rewards as follows:

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots$$

The state *before* performing action a_t is s_t, and the reward *after* action a_t is r_t. The reward r_t only corresponds to the specific amount received at time t, and we need a notion of *cumulative future reward over the long term* for each state-action pair in order to estimate its inherent value. The cumulative expected reward $E[R_t|s_t, a_t]$ for state-action pair (s_t, a_t) is given by the discounted sum of all future expected rewards at discount factor $\gamma \in (0, 1)$:

$$E[R_t|s_t, a_t] = E[r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} \ldots |s_t, a_t] = \sum_{i=0}^{\infty} \gamma^i E[r_{t+i}|s_t, a_t] \quad (11.2)$$

The discount factor $\gamma \in (0, 1)$ regulates how myopic we want to be in allocating rewards. The value of γ is an application-specific parameter less than 1, because future rewards are considered less important than immediate rewards. Choosing $\gamma = 0$ will result in myopically setting the full reward R_t to r_t and nothing else. Larger values of γ will have a better long-term perspective but will need more data for robust learning. *If the cumulative expected rewards of all state-action pairs can be learned, it provides a basis for a reinforcement learning algorithm of selecting the best action in each state.* There are, however, numerous challenges in learning the values of state-action pairs for the following reasons:

- If we define a state-sensitive exploration-exploitation *policy* (like ϵ-greedy in multiarmed bandits) in order to create randomized action sequences for estimating $E[R_t|s_t, a_t]$, the estimated value of $E[R_t|s_t, a_t]$ will be sensitive to the policy used. This is because actions have long-lasting consequences, which can interact with subsequent choices of the policy. For example, a heavily exploratory policy will not learn a large value for $E[R_t|s_t, a_t]$ for those actions a_t that bring a robot near the edge of a

cliff, even if a_t is an optimal action in state s_t in a non-exploratory setting. It is customary to use the notation $E^p[R_t|s_t, a_t]$ to show that the expected values are specific to a policy p. Fortunately, $E^p[R_t|s_t, a_t]$ is still quite helpful in predicting high-quality actions for reasonable policies.

- The reinforcement learning system might have a very large number of states (such as the number of positions in chess). Therefore, explicit tabulation of $E^p[R_t|s_t, a_t]$ is no longer possible, and it needs to learned as a parameterized *function* of (s_t, a_t). This task of model generalization is the primary function of deep learning.

We first introduce the simplest approach, referred to as Monte Carlo sampling.

11.4 Monte Carlo Sampling

The simplest method for reinforcement learning is to use Monte Carlo sampling in which sequences of actions are sampled using a policy p that exploits currently estimated values of $E^p[R_t|s_t, a_t]$, while improving these estimates as well. Each sampled sequence (episode) is referred to as a *Monte Carlo rollout*. This approach is a generalization of the multi-armed bandit algorithm, in which different arms are sampled repeatedly to learn the values of actions. In stateless multi-armed bandits, rewards only depend on actions. In general reinforcement learning, we also have a notion of state, and therefore, one must estimate $E^p[R_t|s_t, a_t]$ for state-action *pairs*, rather than simply actions. Therefore, the key idea is to apply a generic randomized sampling policy, such as the ϵ-greedy policy, in a state-sensitive way. In the following section, we will introduce a simple Monte Carlo algorithm with the ϵ-greedy policy, although one can design similar algorithms with other policies such as a randomized variant of the upper-bounding policy or biased sampling. *Note that the learned values of state-action pairs are sensitive to the policy used or even the value of* ϵ.

11.4.1 Monte Carlo Sampling Algorithm

The ϵ-greedy algorithm for multi-armed bandits is the simplest example of a Monte Carlo sampling algorithm, where one is simulating slot machines to decide which actions are rewarding. Even for deterministic games, this policy is inherently randomized as long as $\epsilon > 0$. In this section, we will show how one can generalize the stateless ϵ-greedy algorithm to a setting with states (using the game of tic-tac-toe as an example). Recall that the gambler (of multiarmed bandits) continuously tries different arms of the slot machine in order to learn more profitable actions in the longer term. However, the tic-tac-toe environment is no longer stateless, and the choice of move depends on the current state of the tic-tac-toe board. In this case, each board position is a state, and the action corresponds to placing 'X' or 'O' at a valid position. The number of valid states of the 3×3 board is bounded above by $3^9 = 19683$, which corresponds to three possibilities ('X', 'O', and blank) for each of 9 positions. Note that all these 19683 positions may not be valid, and therefore they may not be reached in an actual game.

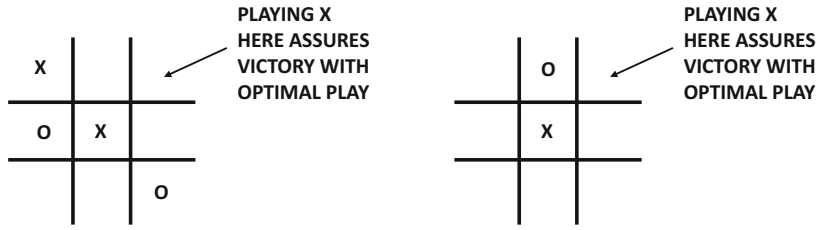
Instead of estimating the value of each (stateless) action in multi-armed bandits, we now estimate the value of each state-action *pair* (s, a) based on the historical performance of action a in state s against a human opponent who is assumed to be the 'O' player (while the agent plays 'X'). Shorter wins are preferred at discount factor $\gamma < 1$, and therefore the *unnormalized* value of action a in state s is increased with γ^{r-1} in case of wins and $-\gamma^{r-1}$ in case of losses after r moves (including the current move). Draws are credited with 0. The

normalized values, which represent estimates of $E^p[R_t|s_t, a_t]$, are obtained by dividing the unnormalized values with the number of times the state-action pair was updated (which is maintained separately). The table starts with small random values, and the action a in state s is chosen greedily to be the action with the highest normalized value with probability $1-\epsilon$, and is chosen to be a random action otherwise. All moves in a game are credited after the termination of each game. In settings where rewards are given after each action, it is necessary to update all past actions in the rolled out episode with time-discounted values. Over time, the values of all state-action pairs for the 'X' player will be learned and the resulting moves will also adapt to the play of the human opponent. Using human opponents slows training. Therefore, one can use self-play to generate these tables optimally without human effort. When self-play is used, separate tables are maintained for the 'X' and 'O' players. The tables are updated from a value in $\{-\gamma^r, 0, \gamma^r\}$ depending on win/draw/loss *from the perspective of the player for whom moves are made*. Over many rollouts, the value of ϵ is often annealed to 0. At inference time, the move with the highest normalized value from either the 'X' table or the 'O' table is selected (while setting $\epsilon = 0$).

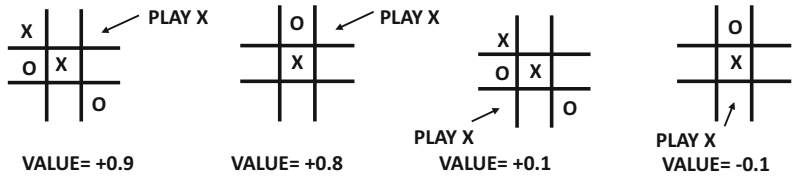
The overarching goal of the Monte Carlo sampling algorithm for tic-tac-toe is to learn the inherent *long-term* value $E^p[R_t|s_t, a_t]$ of each state-action pair. By playing through each game, some state-action pairs are more likely to win than others, and this fact will eventually be reflected in the tabular statistics of state-action pairs. In the early phases of training, only state-action pairs that are very close to win/loss/draws (i.e., one or two moves lead to the outcome), will have accurate long-term values, whereas the early positions on the board will not have accurate values. However, as these statistics improve with more Monte Carlo rollouts, the values of early-stage positions also start becoming increasingly accurate. As a result, the training process is able to learn which actions have the best long-term outcomes in a particular state. For example, making a clever move in tic-tac-toe might set a trap, which eventually results in assured victory. Examples of two such scenarios are shown in Figure 11.2(a) (although the trap on the right is somewhat less obvious). Therefore, one needs to credit a *strategically* good move favorably in the table of state-action pairs and not just the final winning move. The trial-and-error technique based on the Monte Carlo method of section 11.4.1 will indeed assign high values to clever traps. Examples of typical values from such a table are shown in Figure 11.2(b). Note that the less obvious trap of Figure 11.2(a) has a slightly lower value because moves assuring wins after longer periods are discounted by γ, and sampling-based trial-and-error might have a harder time finding the win after setting the trap. This approach can also be use for non-adversarial settings, such as learning to play video games or training robots to walk.

11.4.2 Monte Carlo Rollouts with Function Approximators

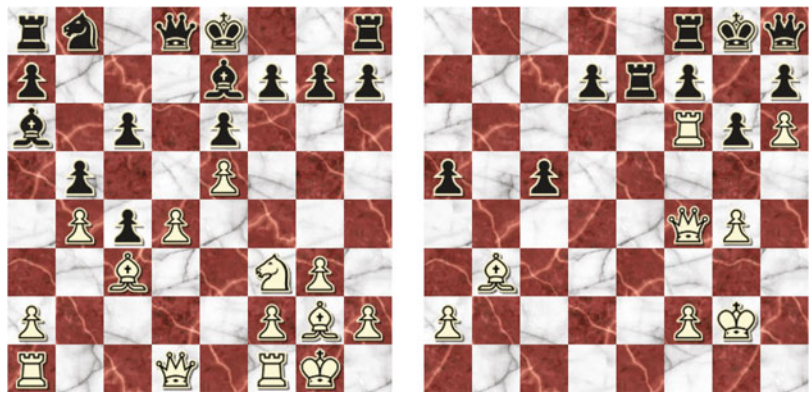
The main problem with this approach is that the number of states in many reinforcement learning settings is too large to tabulate explicitly. For example, the number of possible states in a game of chess is so large that the set of all known positions by humanity is a minuscule fraction of the valid positions. In fact, the algorithm of section 11.4.1 is a refined form of *rote learning* in which Monte Carlo simulations are used to refine and remember the long-term values of *seen* states. One learns about the value of a trap in tic-tac-toe only because previous Monte Carlo simulations have experienced victory many times *from that exact board position*. In most challenging settings like chess, one must *generalize* knowledge learned from prior experiences to a state that the learner has not seen before. All forms of learning (including reinforcement learning) are most useful when they are used to generalize known experiences to unknown situations. In such cases, the table-centric forms



(a) Two examples from tic-tac-toe assuring victory down the road.



(b) Four entries from the table of state-action values in tic-tac-toe. Trial-and-error learns that moves assuring victory have high value.



(c) Positions from two different games between *Alpha Zero* (white) and *Stockfish* (black) [467]: On the left, white sacrifices a pawn and concedes a passed pawn in order to trap black's light-square bishop behind black's own pawns. This strategy eventually resulted in a victory for white after many more moves than the horizon of a conventional chess-playing program like *Stockfish*. In the second game on the right, white has sacrificed material to incrementally cramp black to a position where all moves worsen the position. Incrementally improving positional advantage is the hallmark of the very best human players rather than chess-playing software like *Stockfish*, whose hand-crafted evaluations sometimes fail to accurately capture subtle differences in positions. The neural network in reinforcement learning, which uses the board state as input, evaluates positions in an integrated way without any prior assumptions. The data generated by trial-and-error provides the only experience for training a very complex evaluation function that is indirectly encoded within the parameters of the neural network. The trained network can therefore *generalize* these learned experiences to new positions. This is similar to how humans learn from previous games to better evaluate board positions.

Figure 11.2: Deep learners are needed for large state spaces like (c).

of reinforcement learning are woefully inadequate. Deep learning models serve the role of *function approximators*. Instead of learning and *tabulating* the values of all moves in all positions (using reward-driven trial and error), one learns the value of each position as a *function* of the input state, based on a *trained model* using the outcomes of prior positions. The idea is that the learner can discover *important patterns on the board* and integrate them into an evaluation of a particular position. In other words, *explicit tabulation of the explosive number of possibilities is replaced with a compressed machine learning model*. Without this approach, reinforcement learning cannot be used beyond toy settings like tic-tac-toe.

Consider the design of a Monte Carlo algorithm for chess. We use the same ϵ-greedy algorithm of section 11.4.1, but the value of state-action pair (s_t, a_t) is *estimated* by using the board state after action a_t as input to a convolutional neural network. The output is the estimate of $E^p[R_t|s_t, a_t]$. The ϵ-greedy sampling algorithm is simulated to termination by exploiting the estimated value of $E^p[R_t|s_t, a_t]$. The discounted ground-truth value of each chosen move is drawn from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ based on game outcome and number of moves r to termination. Instead of updating a table of state-action pairs for each chosen move in the simulation, the parameters of the neural network are updated by treating the corresponding board position as a training point together with the ground-truth value from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$. At inference time, the move a_t with the largest $E^p[R_t|s_t, a_t]$ predicted by the neural network can be chosen without the need for ϵ-greedy exploration.

Although the aforementioned approach is somewhat naive, it can be strengthened with Monte Carlo tree search; one such system, known as *Alpha Zero*, has recently been trained [467] to play chess. Two examples of positions [467] from different games in the match between *Alpha Zero* and a conventional chess program, *Stockfish-8.0*, are provided in Figure 11.2(c). In the chess position on the left, the reinforcement learning system makes a *strategically* astute move of cramping the opponent's bishop at the expense of immediate material loss, which most hand-crafted computer evaluations would not prefer. In the position on the right, *Alpha Zero* has sacrificed two pawns and a piece exchange in order to incrementally constrict black to a point where all its pieces are completely paralyzed. Even though *AlphaZero* (probably) never encountered these specific positions during training, its deep learner has the ability to extract relevant features and patterns from previous trial-and-error experience in other board positions. In this particular case, the neural network seems to recognize the primacy of spatial patterns representing subtle positional factors over tangible material factors (much like a human's neural network).

In real-world settings, states are often described using sensory inputs. The deep learner uses this input representation of the state to learn the values of specific actions (e.g., making a move in a game) in lieu of the table of state-action pairs. Even when the input representation of the state (e.g., pixels) is quite primitive, neural networks are masters at squeezing out the relevant insights. This is similar to the approach used by humans to process primitive sensory inputs to define the *state* of the world and make decisions about *actions* using our biological neural network. We do not have a table of pre-memorized state-action pairs for every possible real-life situation. The deep-learning paradigm converts the forbiddingly large table of state-action values into a parameterized model mapping state-action pairs to values, which can be trained easily with backpropagation.

11.5 Bootstrapping for Value Function Learning

The Monte Carlo sampling approach does not work for *non-episodic settings*. In episodic settings like tic-tac-toe, a fixed-length sequence of at most nine moves can be used to

characterize the full and final reward. In non-episodic settings like robots, one is forced to assign credit to an infinitely long sequence in the past based on the reward. Creating a sample of the ground-truth reward by Monte Carlo sampling also has *high variance* because a particular outcome is a very noisy estimate of what might happen *in expectation*. The noise increases with the length of the episode, which increases the number of required rollouts. In other words, we need a methodology for reducing randomness by sampling only a small number of actions. Unfortunately, after a small number of actions, one will not reach a terminal state, which is required for final value estimation (i.e., drawing a sample of $E[R_t|s_t, a_t]$). However, it is possible to estimate *improved* values of states with the methodology of *bootstrapping*, which makes the assumption that *states further along the decision process always have existing value estimates that are better than those states that are earlier along the decision process*. Therefore, one can use the current estimate of the value of the non-terminal state after a few actions in order to better estimate the value of the current state. The idea of bootstrapping may, therefore, be summarized as follows:

Intuition 11.5.1 (Bootstrapping) *Consider a Markov decision process in which we have running estimates of the values (e.g., long-term rewards) of states. We can use a partial simulation of the future to improve the value estimation of the state at the current time-stamp by adding the discounted rewards over these simulated actions with the discounted value of the state reached at the end of the simulation.*

Samuel's checkers program [438] used the running estimate of the value of a position obtained by looking several moves ahead in order to update to an improved estimation of the current position. The idea is that the estimate obtained from looking ahead at the game position after performing the optimal move for each opponent (causing a state of the game closer to the end) is more robust than the estimate of the current position and can therefore be used to improve the value estimate of the current position.

11.5.1 Q-Learning

Consider a Markov decision process with the following sequence of states, actions, and rewards denoted by the repeating sequence $s_t a_t r_t$ at time stamp t. It is assumed that rewards are earned with discount factor γ based in Equation 11.2. The *Q-function* or *Q-value* for the state-action pair (s_t, a_t) is denoted by $Q(s_t, a_t)$, and is a measure of the *inherent* (i.e., long-term) value of performing the action a_t in state s_t (under the *best* possible choice of actions). This value can be viewed as an optimized version $E^*[R_t|s_t, a_t]$ of the expected reward in Equation 11.2. One can choose the next action of the agent by maximizing this value over the set A of all possible actions:

$$a_t^* = \text{argmax}_{a_t \in A} Q(s_t, a_t) \tag{11.3}$$

This predicted action is a good choice for the next move, although it is often combined with an exploratory component (e.g., ϵ-greedy policy) to improve long-term training outcomes. This is exactly similar to how actions are chosen in Monte carlo rollouts. The main difference is in terms of how $Q(s_t, a_t)$ is computed with bootstrapping rather than rollouts and in terms of finding an *optimal* policy.

Instead of using explicit rollouts to termination for crediting the values of state-value pairs, one computes $Q(s_t, a_t)$ by using a single step to s_{t+1}, and then using the best possible Q-value estimate at state s_{t+1} in order to update $Q(s_t, a_t)$. This type of update is referred to as *Bellman's equation*, which is a form of dynamic programming:

$$Q(s_t, a_t) \Leftarrow r_t + \gamma \text{max}_a Q(s_{t+1}, a) \tag{11.4}$$

This update is based on the bootstrapping principle of Intuition 11.5.1, where we are using the better running estimates of states one step ahead in order to create an improved estimate of $Q(s_t, a_t)$. *The above update of Equation 11.4 replaces the updates of Monte Carlo sampling, in order to create the Q-learning algorithm.* While one is using the actual outcomes resulting from a rollout in Monte Carlo sampling, the Q-learning approach is approximating the best possible outcome with a combination of bootstrapping and dynamic programming. It is tempting to continue the simulation of the state-space sampling (like in Monte Carlo sampling). However, an important point is that the *best possible action is not used for making steps*. Rather, the best possible move is made with probability $(1 - \epsilon)$ and a random move is made with probability ϵ in order to move to the next iteration of the algorithm. This is again done in order to be able to navigate the exploration-exploitation trade-off properly. The simulation is continued in order to improve the tabular estimates of $Q(s, a)$ over time. Unlike Monte Carlo sampling, there is a dichotomy between how steps are made (with randomized exploration) and how updates to $Q(s, a)$ are made (in an optimal way with dynamic programming). This dichotomy means that no matter what policy (e.g., ϵ-greedy or biased sampling) is used for simulations, one will always compute the same value of $E^*[R_t|s_t, a_t] = Q(s_t, a_t)$, which is the optimal policy.

In the case of episodic sequences, the Monte Carlo sampling method of the previous section improves the estimates of the values of state-action pairs that are later in the sequence (and closer to termination) first. This is also the case in Q-learning, where the updates for a state that is one step from termination is exact. It is important to set $\hat{Q}(s_{t+1}, a)$ to 0 in case the process terminates after performing a_t for episodic sequences. Therefore, the accuracy of the estimation of the value of the state-action pair will be propagated over time from states closer to termination to earlier states (which is the fundamental principle of bootstrapping). For example, in a tic-tac-toe game, the value of a winning action in a state will be accurately estimated in a single iteration, whereas the value of an early move in tic-tac-toe will require the propagation of values from later states to earlier states via the Bellman equation. This type of propagation will require a few iterations.

In cases where the state-space is very small (like tic-tac-toe), one can learn $Q(s_t, a_t)$ explicitly by using the Bellman equations (cf. Equation 11.4) at each move to update an *array* containing the explicit value of $Q(s_t, a_t)$. However, using Equation 11.4 directly might result in instability early on because of the random initialization of entries. Therefore, gentle updates are performed using the learning rate $\alpha < 1$:

$$Q(s_t, a_t) \Leftarrow Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a)) \tag{11.5}$$

Using $\alpha = 1$ will result in Equation 11.4. Updating the array continually will result in a table containing the correct *strategic* value of each move; see, for example, Figure 11.2(a) for an understanding of the notion of strategic value. Figure 11.2(b) contains examples of four entries from such a table. Note that this is a direct alternative to the tabular approach to Monte Carlo sampling and it will result in the same values as in the final table as Monte Carlo sampling, but it will work only for toy settings like tic-tac-toe.

11.5.2 Deep Learning Models as Function Approximators

As in the case of Monte Carlo sampling, it is impossible to tabulate (or evaluate even once) the values of all possible states in large state spaces like chess or video games. This is where settings like chess and video games differ from tic-tac-toe. Note that Equation 11.5 works only when the estimates on the right-hand side are meaningful. If the state $Q(s_{t_1}, a)$ on the right-hand side of Equation 11.5 has never been encountered most of the time (even after a

long period of learning), the updates will be useless. Therefore, function approximators are needed (as in Monte Carlo sampling).

For ease in discussion, we will work with the Atari setting [346] in which a fixed window of the last few snapshots of pixels provides the state s_t. Assume that the feature representation of s_t is denoted by $\overline{X}_t$. The neural network uses $\overline{X}_t$ as the input and outputs $Q(s_t, a)$ for each possible legal action a from the universe of actions denoted by the set A.

Assume that the neural network is parameterized by the column vector of weights $\overline{W}$, and it has $|A|$ outputs containing the Q-values corresponding to the various actions in A. In other words, for each action $a \in A$, the neural network is able to compute the function $F(\overline{X}_t, \overline{W}, a)$, which is defined to be the *learned estimate* of $Q(s_t, a)$:

$$F(\overline{X}_t, \overline{W}, a) = \hat{Q}(s_t, a) \tag{11.6}$$

Note the circumflex on top of the Q-function in order to indicate that it is a predicted value using the learned parameters $\overline{W}$. Learning $\overline{W}$ is the key to using the model for deciding which action to use at a particular time-stamp. For example, consider a video game in which the possible moves are up, down, left, and right. In such a case, the neural network will have four outputs as shown in Figure 11.3. In the specific case of the Atari 2600 games, the input contains $m = 4$ spatial pixel maps in grayscale, representing the window of the last m moves [346, 347]. A convolutional neural network is used to convert pixels into Q-values. This network is referred to as a *Q-network*. We will provide more details of the specifics of the architecture later.

The weights $\overline{W}$ of the neural network need to be learned via training. Here, we encounter an interesting problem. We can learn the vector of weights only if we have *observed* values of the Q-function. With observed values of the Q-function, we could easily set up a loss in terms of $Q(s_t, a) - \hat{Q}(s_t, a)$ in order to perform the learning after each action. The problem is that the Q-function represents the maximum discounted reward over all *future* combinations of actions, and there is no way of observing it at the current time.

It is here that the bootstrapping trick is used for setting up the neural network loss function. According to Intuition 11.5.1, *we do not really need the observed Q-values in order to set up a loss function as long as we know an improved estimate of the Q-values by using partial knowledge from the future.* Then, we can use this improved estimate to create a surrogate "observed" value. This "observed" value is defined by the Bellman equation (cf. Equation 11.4) discussed earlier:

$$Q(s_t, a_t) = r_t + \gamma \max_a \hat{Q}(s_{t+1}, a) \tag{11.7}$$

The correctness of this relationship follows from the fact that the Q-function is designed to maximize the discounted future payoff. We are essentially looking at all actions one step ahead in order to create an improved estimate of $Q(s_t, a_t)$. It is important to set $\hat{Q}(s_{t+1}, a)$ to 0 in case the process terminates after performing a_t for episodic sequences. We can write this relationship in terms of our neural network predictions as well:

$$F(\overline{X}_t, \overline{W}, a_t) = r_t + \gamma \max_a F(\overline{X}_{t+1}, \overline{W}, a) \tag{11.8}$$

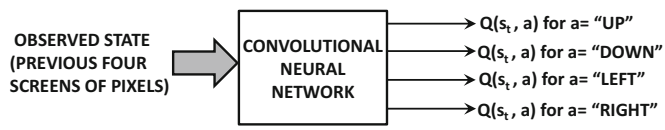


Figure 11.3: The Q-Network for the Atari video game setting

Note that one must first wait to observe the state $\overline{X}_{t+1}$ and reward r_t by performing the action a_t, before we can compute the "observed" value at time-stamp t on the right-hand side of the above equation. This provides a natural way to express the loss L_t of the neural network at time stamp t by comparing the (surrogate) observed value to the predicted value at time stamp t:

$$L_t = \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2 \tag{11.9}$$

Therefore, we can now update the vector of weights $\overline{W}$ using backpropagation on this loss function. Here, it is important to note that the target values estimated using the inputs at $(t+1)$ are treated as constant ground-truths by the backpropagation algorithm. Therefore, the derivative of the loss function will treat these estimated values as constants, even though they were obtained from the parameterized neural network with input $\overline{X}_{t+1}$. Not treating $F(\overline{X}_{t+1}, \overline{W}, a)$ as a constant will lead to poor results. This is because we are treating the prediction at $(t+1)$ as an improved estimate of the ground-truth (based on the bootstrapping principle). Therefore, the backpropagation algorithm will compute the following:

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{11.10}$$

In matrix-calculus notation, the partial derivative of a function $F()$ with respect to the vector $\overline{W}$ is essentially the gradient $\nabla_{\overline{W}} F$. At the beginning of the process, the Q-values estimated by the neural network are random because the vector of weights $\overline{W}$ is initialized randomly. However, the estimation gradually becomes more accurate with time, as the weights are constantly changed to maximize rewards.

Therefore, at any given time-stamp t at which action a_t and reward r_t has been observed, the following training process is used for updating the weights $\overline{W}$:

1. Perform a forward pass through the network with input $\overline{X}_{t+1}$ to compute $\hat{Q}_{t+1} = \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)$. The value is 0 in case of termination after performing a_t. *Treating the terminal state specially is important.* According to the Bellman equations, the Q-value at previous time-stamp t should be $r_t + \gamma \hat{Q}_{t+1}$ for observed action a_t at time t. Therefore, instead of using observed values of the target, we have created a *surrogate* for the target value at time t, and we pretend that this surrogate is an observed value given to us.

2. Perform a forward pass through the network with input $\overline{X}_t$ to compute $F(\overline{X}_t, \overline{W}, a_t)$.

3. Set up a loss function in $L_t = (r_t + \gamma Q_{t+1} - F(\overline{X}_t, \overline{W}, a_t))^2$, and backpropagate in the network with input $\overline{X}_t$. Note that this loss is associated with neural network output node corresponding to action a_t, and the loss for all other actions is 0.

4. One can now use backpropagation on this loss function in order to update the weight vector $\overline{W}$. Even though the term $r_t + \gamma Q_{t+1}$ in the loss function is also obtained as a prediction from input $\overline{X}_{t+1}$ to the neural network, it is treated as a (constant) observed value during gradient computation by the backpropagation algorithm.

Both the training and the prediction are performed simultaneously, as the values of actions are used to update the weights and select the next action. It is tempting to select the action with the largest Q-value as the relevant prediction. However, such an approach might perform inadequate exploration of the search space. Therefore, one couples the optimality prediction with a policy such as the ϵ-greedy algorithm in order to select the next action. The action with the largest predicted payoff is selected with probability $(1 - \epsilon)$. Otherwise, a random action is selected. The value of ϵ can be annealed by starting with large values and reducing them over time. Therefore, the *target prediction value* for the neural network is computed using the best possible action in the Bellman equation (which might eventually be different from observed action a_{t+1} based on the ϵ-greedy policy). This is the reason that Q-learning is referred to as an *off-policy algorithm* in which the target prediction values for the neural network update are computed using actions that might be different from the actually observed actions in the future.

11.5.3 Example: Neural Network Specifics for Video Game Setting

For the convolutional neural network [346, 347], the screen sizes were set to 84×84 pixels, which also defined the spatial footprints of the first layer in the convolutional network. The input was in grayscale, and therefore each screen required only a single spatial feature map, although a depth of 4 was required in the input layer to represent the previous four windows of pixels. Three convolutional layers were used with filters of size 8×8, 4×4, and 3×3, respectively. A total of 32 filters were used in the first convolutional layer, and 64 filters were used in each of the other two, with the strides used for convolution being 4, 2, and 1, respectively. The convolutional layers were followed by two fully connected layers. The number of neurons in the penultimate layer was equal to 512, and that in the final layer was equal to the number of outputs (possible actions). The number of output layers varied between 4 and 18, and was game-specific. The overall architecture of the convolutional network is illustrated in Figure 11.4.

All hidden layers used the ReLU activation, and the output used linear activation in order to predict the real-valued Q-value. No pooling was used, and the strides in the convolution provided spatial compression. The Atari platform supports many games, and the same broader architecture was used across different games in order to showcase its generalizability. There was some variation in performance across different games, although human performance was exceeded in many cases. The algorithm faced the greatest challenges in games in which longer-term strategies were required. Nevertheless, the robust performance of a relatively homogeneous framework across many games was encouraging.

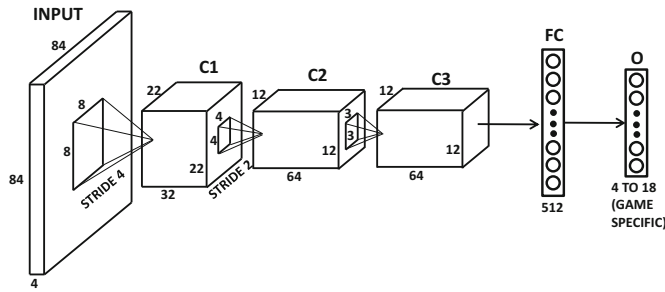


Figure 11.4: The convolutional neural network for the Atari setting

11.5.4 On-Policy versus Off-Policy Methods: SARSA

The Q-Learning methodology belongs to the class of methods, referred to as *temporal difference learning*. In Q-learning, the actions are chosen according to an ϵ-greedy policy. However, the parameters of the neural network are updated based on the best possible action at each step with the Bellman equation. The best possible action at each step is not quite the same as the ϵ-greedy policy used to perform the simulation. Therefore, Q-learning is an *off-policy reinforcement learning method*. Choosing a different policy for executing actions from those for performing updates is a consequence of the fact that the Bellman updates are intended to find an *optimal policy* rather than *evaluating* a specific policy like ϵ-greedy (as is the case with Monte Carlo methods). In *on-policy methods* like Monte Carlo sampling, the actions are consistent with the updates, and therefore the updates can be viewed as policy *evaluation* rather than policy *optimization*. Therefore, changing the policy affects the predicted actions more significantly in Monte Carlo sampling than in Q-Learning. A bootstrapped approximation of Monte Carlo sampling can also be achieved with the use of the SARSA (State-Action-Reward-State-Action) algorithm, in which the reward in the next step is updated using the action a_{t+1} predicted by the ϵ-greedy policy rather than the optimal step from the Bellman equation. Let $Q^p(s, a)$ be the evaluation of policy p (which is ϵ-greedy in this case) for state-action pair (s, a). Then, after sampling action a_{t+1} using ϵ-greedy, the update is as follows:

$$Q^p(s_t, a_t) \Leftarrow Q^p(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma Q^p(s_{t+1}, a_{t+1})) \tag{11.11}$$

If action a_t at state s_t leads to termination (for episodic processes), then $Q^p(s_t, a_t)$ is simply set to r_t. Note that this update is different from the Q-Learning update of Equation 11.5, because action a_{t+1} includes the effect of exploration.

When using function approximators, the loss function for the next step is defined as follows:

$$L_t = \left\{r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) - F(\overline{X}_t, \overline{W}, a_t)\right\}^2 \tag{11.12}$$

The function $F(\cdot, \cdot, \cdot)$ is defined in the same way as the previous section. The weight vector is updated based on this loss, and then the action a_{t+1} is executed:

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{11.13}$$

Here, it is instructive to compare this update with those used in Q-learning according to Equation 11.10. In Q-learning, one is using the *best possible* action at each state in order to update the parameters, even though the policy that is actually executed might be ϵ-greedy (which encourages exploration). In SARSA, we are using the action that was actually selected by the ϵ-greedy method in order to perform the update. Therefore, the approach is an *on-policy method*. Off-policy methods like Q-learning are able to decouple exploration from exploitation, whereas on-policy methods are not. Note that if we set the value of ϵ in the ϵ-greedy policy to 0 (i.e., vanilla greedy), then both Q-Learning and SARSA would specialize to the same algorithm. However, such an approach would not work very well because there is no exploration. SARSA is useful when learning cannot be done separately from prediction. Q-learning is useful when the learning can to be done offline, which is followed by exploitation of the learned policy with a vanilla-greedy method at $\epsilon = 0$ (and no need for further model updates). Using ϵ-greedy at inference time would be dangerous

in Q-learning, because the policy never pays for its exploratory component (in the update) and therefore does not learn how to keep exploration safe. For example, a Q-learning based robot will take the shortest path to get from point A to point B even if it is along the edge of the cliff, whereas a SARSA-trained robot will not. SARSA can also be implemented with n-step lookaheads (rather than 1-step bootstrapping), which brings it even closer to the Monte Carlo sampling of section 11.4.

11.5.5 Modeling States versus State-Action Pairs

A minor variation of the theme in the previous sections is to learn the value of a particular state (rather than state-action pair). One can implement all the methods discussed earlier by maintaining values of states rather than state-action pairs. For example, SARSA can be implemented by evaluating all the values of states resulting from each possible action and selecting a good one based on a pre-defined policy like ϵ-greedy. In fact, the earliest methods for temporal difference learning (or *TD-learning*) maintained values on states rather than state-action pairs. From an efficiency perspective, it is more convenient to output the values of all actions in one shot (rather than repeatedly evaluate each forward state) for value-based decision making. Working with state values rather that state-action pairs becomes useful only when the policy cannot be expressed neatly in terms of state-action pairs. For example, we might evaluate a forward-looking tree of promising moves in chess, and report some averaged value for bootstrapping. In such cases, it is desirable to evaluate states rather than state-action pairs. This section will therefore discuss a variation of temporal difference learning in which states are directly evaluated.

Let the value of the state s_t be denoted by $V(s_t)$. Now assume that you have a parameterized neural network that uses the observed attributes $\overline{X}_t$ (e.g., pixels of last four screens in Atari game) of state s_t to estimate $V(s_t)$. An example of this neural network is shown in Figure 11.5. Then, if the function computed by the neural network is $G(\overline{X}_t, \overline{W})$ with parameter vector $\overline{W}$, we have the following:

$$G(\overline{X}_t, \overline{W}) = \hat{V}(s_t) \tag{11.14}$$

Note that the policy being followed to decide the actions might use some arbitrary evaluation of forward-looking states to decide actions. For now, we will assume that we have some reasonable heuristic policy for choosing the actions that uses the forward-looking state values in some way. For example, if we evaluate each forward state resulting from an action and select one of them based on a pre-defined policy (e.g., ϵ-greedy), the approach discussed below is the same as SARSA.

If the action a_t is performed with reward r_t, the resulting state is s_{t+1} with value $V(s_{t+1})$. Therefore, the bootstrapped ground-truth estimate for $V(s_t)$ can be obtained with the help of this lookahead:

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{11.15}$$

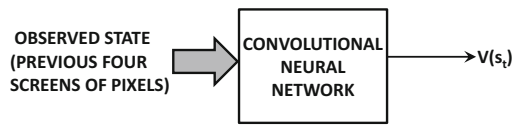


Figure 11.5: Estimating the value of a state with temporal difference learning

This estimate can also be stated in terms of the neural network parameters:

$$G(\overline{X}_t, \overline{W}) = r_t + \gamma G(\overline{X}_{t+1}, \overline{W}) \tag{11.16}$$

During the training phase, one needs to shift the weights so as to push $G(\overline{X}_t, \overline{W})$ towards the improved "ground truth" value of $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$. As in the case of Q-learning, we work with the boot-strapping pretension that the value $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$ is an observed value given to us. Therefore, we want to minimize the *TD-error* defined by the following:

$$\delta_t = \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \tag{11.17}$$

Therefore, the loss function L_t is defined as follows:

$$L_t = \delta_t^2 = \left\{ \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \right\}^2 \tag{11.18}$$

As in Q-learning, one would first compute the "observed" value of the state at time stamp t using the input $\overline{X}_{t+1}$ into the neural network to compute $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$. Therefore, one would have to wait till the action a_t has been observed, and therefore the observed features $\overline{X}_{t+1}$ of state s_{t+1} are available. This "observed" value (defined by $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$) of state s_t is then used as the (constant) target to update the weights of the neural network, when the input $\overline{X}_t$ is used to predict the value of the state s_t. Therefore, one would need to move the weights of the neural network based on the gradient of the following loss function:

$$\begin{aligned}
\overline{W} &\Leftarrow \overline{W} - \alpha \frac{\partial L_t}{\partial \overline{W}} \\
&= \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma G(\overline{X}_{t+1}, \overline{W})]}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \right\} \frac{\partial G(\overline{X}_t, \overline{W})}{\partial \overline{W}} \\
&= \overline{W} + \alpha \delta_t (\nabla G(\overline{X}_t, \overline{W}))
\end{aligned}$$

This algorithm is a special case of the $TD(\lambda)$ algorithm with λ set to 0. This special case only updates the neural network by creating a bootstrapped "ground-truth" for the current time-stamp based on the evaluations of the next time-stamp. This type of ground-truth is an inherently myopic *approximation*. For example, in a chess game, the reinforcement learning system might have inadvertently made some mistake many steps ago, and it is suddenly showing high errors in the bootstrapped predictions without having shown up earlier. The errors in the bootstrapped predictions are indicative of the fact that we have received new information about each past state $\overline{X}_k$, which we can use to alter its prediction. One possibility is to bootstrap by looking ahead for multiple steps (see Exercise 7). Another solution is the use of $TD(\lambda)$, which explores the continuum between perfect Monte Carlo ground truth and single-step approximation with smooth decay. The adjustments to older predictions are increasingly discounted at the rate $\lambda < 1$. In such a case, the update can be shown to be the following [498]:

$$\overline{W} \Leftarrow \overline{W} + \alpha \delta_t \sum_{k=0}^{t} \underbrace{(\lambda\gamma)^{t-k} (\nabla G(\overline{X}_k, \overline{W}))}_{\text{Alter prediction of } \overline{X}_k} \tag{11.19}$$

At $\lambda = 1$, the approach can be shown to be equivalent to a method in which Monte-Carlo evaluations (i.e., rolling out an episodic process to the end) are used to compute the ground-truth [498]. This is because we are always using new information about errors to fully correct our past mistakes without discount at $\lambda = 1$, thereby creating an unbiased estimate. Note that λ is only used for discounting the steps, whereas γ is also used in computing the TD-error δ_t according to Equation 11.17. The parameter λ is *algorithm-specific*, whereas γ is *environment-specific*. Using $\lambda = 1$ or Monte Carlo sampling leads to lower bias and higher variance. For example, consider a chess game in which agents Alice and Bob each make three errors in a single game but Alice wins in the end. This single Monte Carlo rollout will not be able to distinguish the impact of each specific error and will assign the discounted credit for final game outcome to each board position. On the other hand, an n-step temporal difference method (i.e., n-ply board evaluation) might see a temporal difference error for each board position in which the agent made a mistake and was detected by the n-step lookahead. It is only with sufficient data (i.e., more games) that the Monte Carlo method will distinguish between different types of errors. However, choosing very small values of λ will have difficulty in learning openings (i.e., greater bias) because errors with long-term consequences will not be detected. Such problems with openings are well documented [23, 513].

Temporal difference learning was used in Samuel's celebrated checkers program [438], and also motivated the development of TD-Gammon for Backgammon by Tesauro [509]. A neural network was used for state value estimation, and its parameters were updated using temporal-difference bootstrapping over successive moves. The final inference was performed with minimax evaluation of the improved evaluation function over a shallow depth such as 2 or 3. TD-Gammon was able to defeat several expert players. It also exhibited some unusual strategies of game play that were eventually adopted by top-level players.

11.6 Policy Gradient Methods

The value-based methods like Q-learning attempt to predict the value of an action with the neural network and couple it with a generic policy (like ϵ-greedy). On the other hand, policy gradient methods estimate the *probability* of each action at each step with the goal of maximizing the overall reward. Therefore, the policy is itself parameterized, rather than using the value estimation as an intermediate step for choosing actions.

The neural network for estimating the policy is referred to as a *policy network* in which the input is the current state of the system, and the output is a set of probabilities associated with the various actions in the video game (e.g., moving up, down, left, or right). As in the case of the Q-network, the input can be an observed representation of the agent state. For example, in the Atari video game setting, the observed state can be the last four screens of pixels. An example of a policy network is shown in Figure 11.6, which is relevant for

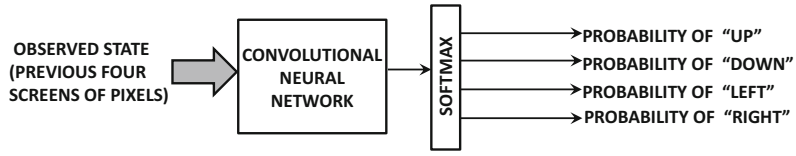


Figure 11.6: The policy network for the Atari video game setting. It is instructive to compare this configuration with the Q-network of Figure 11.3.

the Atari setting. It is instructive to compare this policy network with the Q-network of Figure 11.3. Given an output of probabilities for various actions, we throw a biased die with the faces associated with these probabilities, and select one of these actions. Therefore, for each action a, observed state representation $\overline{X}_t$, and current parameter $\overline{W}$, the neural network is able to compute the function $P(\overline{X}_t, \overline{W}, a)$, which is the probability that the action a should be performed. One of the actions is sampled, and a reward is observed for that action (sometimes using Monte Carlo sampling). If the policy is poor, the action will more likely to be a mistake and the reward will be poor as well. The weight vector $\overline{W}$ is then updated for the next iteration by gradient ascent (in order to *maximize* expected rewards rather than minimize loss). The main challenge is in computing the gradient of an expectation. Common policy gradients methods include *finite difference methods*, *likelihood ratio methods*, and *natural policy gradients*. In the following, we will only discuss the first two methods.

11.6.1 Finite Difference Methods

The method of finite differences side-steps the problem of stochasticity with empirical simulations that provide estimates of the gradient. Finite difference methods use weight perturbations in order to estimate gradients of the reward. The idea is to use s different perturbations of the neural network weights, and examine the expected change ΔJ in the reward. Note that this will require us to run the perturbed policy for the horizon of H moves in order to estimate the change in reward. Such a sequence of H moves is referred to as a *roll-out*. For example, in the case of the Atari game, we will need to play it for a trajectory of H moves for each of these s different sets of perturbed weights in order to estimate the changed reward. In games where an opponent of sufficient strength is not available to train against, it is possible to play a game against a version of the opponent based on parameters learned a few iterations back.

The discounted rewards over a given horizon H are computed using a truncated version of Equation 11.2 over horizon H starting from the very beginning:

$$J = \sum_{t=0}^{H} \gamma^t r_t \tag{11.20}$$

To maximize rewards, we update the weight vector with gradient ascent as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha \nabla J \tag{11.21}$$

In general, the value of H might be large enough that we might reach the end of the game, and therefore the score used will be the one at the end of the game. In some games like *Go*, the score is available only at the end of the game, with a $+1$ for a win and -1 for a loss. In such cases, it becomes more important to choose H large enough so as to play till the end of the game. As a result, we will have s different column vectors $\Delta\overline{W}_1 \ldots \Delta\overline{W}_s$ of weight change, together with corresponding changes $\Delta J_1 \ldots \Delta J_s$ in the total reward. Each of these pairs roughly satisfies the following relationship:

$$(\Delta\overline{W}_r) \cdot \nabla J \approx \Delta J_r \quad \forall r \in \{1 \ldots s\} \tag{11.22}$$

We can create an s-dimensional column vector $\overline{y} = [\Delta J_1 \ldots \Delta J_s]^T$ of the changes in the objective function and an $N \times s$ matrix D by stacking the rows $\Delta\overline{W}_r$ on top of each

other, where N is the number of parameters in the neural network. Therefore, we have the following:

$$D[\nabla J] \approx \overline{y} \tag{11.23}$$

Then, the policy gradient is obtained by performing a straightforward linear regression of the change in objective functions with respect to the change in weight vectors. By using the formula for linear regression (cf. section 3.2.2.2 of Chapter 3), we obtain the following:

$$\nabla J = (D^T D)^{-1} D^T \overline{y} \tag{11.24}$$

This gradient is used for the update in Equation 11.21. It is required to run the policy for a sequence of H moves for each of the s samples to estimate the gradients. This process can sometimes be slow.

11.6.2 Likelihood Ratio Methods

Likelihood-ratio methods were proposed by Williams [552] in the context of the REINFORCE algorithm. Consider the case in which we are following the policy with probability vector $\overline{p}$ and we want to maximize $E[Q^p(s, a)]$, which is the long-term expected value of state s and each sampled action a from the neural network. Consider the case in which the probability of action a is $p(a)$ (which is output by the neural network). In such a case, we want to find the gradient of $E[Q^p(s, a)]$ with respect to the weight vector $\overline{W}$ of the neural network for stochastic gradient ascent. Finding the gradient of an expectation from sampled events is non-obvious. However, the log-probability trick allows us to bring the expectation outside the gradient, which is additive over the samples of state-action pairs:

$$\nabla E[Q^p(s, a)] = E[Q^p(s, a) \nabla \log(p(a))] \tag{11.25}$$

We show the proof of the above result in terms of the partial derivative with respect to a single neural network weight w under the assumption that a is a discrete variable:

$$\frac{\partial E[Q^p(s,a)]}{\partial w} = \frac{\partial \left[\sum_a Q^p(s,a) p(a)\right]}{\partial w} = \sum_a Q^p(s,a) \frac{\partial p(a)}{\partial w} = \sum_a Q^p(s,a) \left[\frac{1}{p(a)} \frac{\partial p(a)}{\partial w}\right] p(a)$$
$$= \sum_a Q^p(s,a) \left[\frac{\partial \log(p(a))}{\partial w}\right] p(a) = E\left[Q^p(s,a) \frac{\partial \log(p(a))}{\partial w}\right]$$

The above result can also be shown for the case in which a is a continuous variable (cf. Exercise 1). Continuous actions occur frequently in robotics (e.g., distance to move arm).

It is easy to use this trick for neural network parameter estimation. Each action a sampled by the simulation is associated with the long-term reward $Q^p(s, a)$, which is obtained by Monte Carlo simulation. Based on the relationship above, the gradient of the expected advantage is obtained by multiplying the gradient of the log-probability $\log(p(a))$ of that action (computable from the neural network in Figure 11.6 using backpropagation) with the long-term reward $Q^p(s, a)$ (obtained by Monte Carlo simulation).

Consider a simple game of chess with a win/loss/draw at the end and discount factor γ In this case, the long-term reward of each move is simply obtained as a value from $\{+\gamma^{r-1}, 0, -\gamma^{r-1}\}$, when r moves remain to termination. The value of the reward depends on the final outcome of the game, and number of remaining moves (because of reward discount). Consider a game containing at most H moves. Since multiple roll-outs are used,

we get a whole bunch of training samples for the various input states and corresponding outputs in the neural network. For example, if we ran the simulation for 100 roll-outs, we would get at most $100 \times H$ different samples. Each of these would have a long-term reward drawn from $\{+\gamma^{r-1}, 0, -\gamma^{r-1}\}$. For each of these samples, the reward serves as a weight during a gradient-ascent update of the log-probability of the sampled action.

$$\overline{W} \Leftarrow \overline{W} + Q^p(s, a)\nabla \log(p(a)) \tag{11.26}$$

Here, $p(a)$ is the neural network's output probability of the sampled action. The gradients are computed using backpropagation, and these updates are similar to those in Equation 11.21. This process of sampling and updating is carried through to convergence.

Note that the gradient of the log-probability of the ground-truth class is often used to update softmax classifiers with cross-entropy loss in order to increase the probability of the correct class (which is intuitively similar to the update here). The difference here is that we are weighting the update with the Q-values because we want to push the parameters more aggressively in the direction of highly rewarding actions. One could also use mini-batch gradient ascent over the actions in the sampled roll-outs. Randomly sampling from different roll-outs can be helpful in avoiding the local minima arising from correlations because the successive samples from each roll-out are closely related to one another.

Reducing Variance with Baselines: Although we have used the long-term reward $Q^p(s, a)$ as the quantity to be optimized, it is more common to subtract a baseline value from this quantity in order to obtain its *advantage* (i.e., differential impact of the action over expectation). The baseline is ideally state-specific, but can be a constant as well. In the original work of REINFORCE, a constant baseline was used (which is typically some measure of average long-term reward over all states). Even this type of simple measure can help in speeding up learning because it reduces the probabilities of less-than-average performers and increases the probabilities of more-than-average performers (rather than increasing both at differential rates). A constant choice of baseline does not affect the bias of the procedure, but it reduces the variance. A *state-specific* option for the baseline is the value $V^p(s)$ of the state s immediately *before* sampling action a. Such a choice results in the advantage $(Q^p(s, a) - V^p(s))$ becoming identical to the temporal difference error. This choice makes intuitive sense, because the temporal difference error contains *additional* information about the differential reward of an action beyond what we would know before choosing the action. Discussions on baseline choice may be found in [387, 452].

Consider an example of an Atari game-playing agent, in which a roll-out samples the move UP and output probability of UP was 0.2. Assume that the (constant) baseline is 0.17, and the long-term reward of the action is +1, since the game results in win (and there is no reward discount). Therefore, the score of every action in that roll-out is 0.83 (after subtracting the baseline). Then, the gain associated with all actions (output nodes of the neural network) other than UP at that time-step would be 0, and the gain associated with the output node corresponding to UP would be $0.83 \times \log(0.2)$. One can then backpropagate this gain in order to update the parameters of the neural network.

Adjustment with a state-specific baseline is easy to explain intuitively. Consider the example of a chess game between agents Alice and Bob. If we use a baseline of 0, then each move will only be credited with a reward corresponding to the final result, and the difference between good moves and bad moves will not be evident. In other words, we need to simulate a lot more games to differentiate positions. On the other hand, if we use the value of the state (before performing the action) as the baseline, then the (more refined) temporal difference error is used as the advantage of the action. In such a case, moves

that have greater state-specific impact will be recognized with a higher advantage (within a single game). As a result, fewer games will be required for learning.

Combining Supervised Learning with Policy Gradients

Supervised learning is useful for initializing the weights of the policy network before applying reinforcement learning. For example, in a game of chess, one might have prior examples of expert moves that are already known to be good. In such a case, we simply perform gradient ascent with the same policy network, except that each expert move is assigned the fixed credit of 1 for evaluating the gradient according to Equation 11.25. This problem becomes identical to that of softmax classification, where the goal of the policy network is to predict the same move as the expert. One can sharpen the quality of the training data with some examples of bad moves with a negative credit obtained from computer evaluations. This approach would be considered supervised learning rather than reinforcement learning because we are simply using prior data, and not generating/simulating the data that we learn from (as is common in reinforcement learning). This general idea can be extended to any reinforcement learning setting, where some prior examples of actions and associated rewards are available. Supervised learning is extremely common in these settings for initialization because of the difficultly in obtaining high-quality data in the early stages of the process. Many published works also interleave supervised learning and reinforcement learning in order to achieve greater data efficiency [292].

11.6.3 Actor-Critic Methods

So far, we have discussed methods that are either dominated by *critics* or by *actors* in the following way:

1. The Q-learning and $TD(\lambda)$ methods work with the notion of a value function that is optimized. This value function is a critic, and the policy (e.g., ϵ-greedy) of the actor is directly derived from this critic. Therefore, such methods are considered *critic-only* methods.

2. The policy-gradient methods directly learn the probabilities of the policy actions. The advantages of various actions are often estimated using Monte Carlo sampling. Therefore, these methods are considered *actor-only* methods.

Note that the policy-gradient methods do need to evaluate the advantage of intermediate actions, and this estimation has so far been done with the use of Monte Carlo simulations. However, Monte Carlo simulations are costly and impractical in an online setting. It turns out that one can learn the advantage of intermediate actions using value function methods. As in the previous section, we use the notation $Q^p(s_t, a)$ to denote the value of action a, when the policy p followed by the policy network is used. Therefore, we would now have two coupled neural networks– a policy network and a Q-network. The policy network learns the probabilities of actions, and the Q-network learns the values $Q^p(s_t, a)$ of various actions in order to provide an estimation of the advantage to the policy network. Therefore, the policy network uses $Q^p(s_t, a)$ (with baseline adjustments) to weight its gradient ascent updates. The Q-network is updated using an on-policy update as in SARSA, where the policy is controlled by the policy network (rather than ϵ-greedy). The Q-network, however, does not directly decide the actions as in Q-learning, because the policy decisions are outside its control (beyond its role as a critic). Therefore, the policy network is the actor and the value

network is the critic. To distinguish between the policy network and the Q-network, we will denote the parameter vector of the policy network by $\overline{\Theta}$, and that of the Q-network by $\overline{W}$.

We assume that the state at time stamp t is denoted by s_t, and the observable features of the state input to the neural network are denoted by $\overline{X}_t$. Therefore, we will use s_t and $\overline{X}_t$ interchangeably below. Consider a situation at the tth time-stamp, where the action a_t has been observed after state s_t with reward r_t. Then, the following sequence of steps is applied for the $(t+1)$th step:

1. Sample the action a_{t+1} using the current state of the parameters in the policy network. Note that the current state is s_{t+1} because the action a_t is already observed.

2. Let $F(\overline{X}_t, \overline{W}, a_t) = \hat{Q}^p(s_t, a_t)$ represent the estimated value of $Q^p(s_t, a_t)$ by the Q-network using the observed representation $\overline{X}_t$ of the states and parameters $\overline{W}$. Estimate $Q^p(s_t, a_t)$ and $Q^p(s_{t+1}, a_{t+1})$ using the Q-network. Compute the TD-error δ_t as follows:

$$\begin{aligned}\delta_t &= r_t + \gamma \hat{Q}^p(s_{t+1}, a_{t+1}) - \hat{Q}^p(s_t, a_t) \\ &= r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) - F(\overline{X}_t, \overline{W}, a_t)\end{aligned}$$

3. **[Update policy network parameters]:** Let $P(\overline{X}_t, \overline{\Theta}, a_t)$ be the probability of the action a_t predicted by policy network. Update the parameters of the policy network as follows:

$$\overline{\Theta} \leftarrow \overline{\Theta} + \alpha \hat{Q}^p(s_t, a_t) \nabla_{\Theta} \log(P(\overline{X}_t, \overline{\Theta}, a_t))$$

   Here, α is the learning rate for the policy network and the value of $\hat{Q}^p(s_t, a_t) = F(\overline{X}_t, \overline{W}, a_t)$ is obtained from the Q-network.

4. **[Update Q-Network parameters]:** Update the Q-network parameters as follows:

$$\overline{W} \Leftarrow \overline{W} + \beta \delta_t \nabla_W F(\overline{X}_t, \overline{W}, a_t)$$

   Here, β is the learning rate for the Q-network. A caveat is that the learning rate of the Q-network is generally higher than that of the policy network.

The action a_{t+1} is then executed in order to observe state s_{t+2}, and the value of t is incremented. The next iteration of the approach is executed (by repeating the above steps) at this incremented value of t. The iterations are repeated, so that the approach is executed to convergence. The value of $\hat{Q}^p(s_t, a_t)$ is the same as the value $\hat{V}^p(s_{t+1})$.

If we use $\hat{V}^p(s_t)$ as the baseline, the advantage $\hat{A}^p(s_t, a_t)$ is defined by the following:

$$\hat{A}^p(s_t, a_t) = \hat{Q}^p(s_t, a_t) - \hat{V}^p(s_t)$$

This changes the updates as follows:

$$\overline{\Theta} \leftarrow \overline{\Theta} + \alpha \hat{A}^p(s_t, a_t) \nabla_{\Theta} \log(P(\overline{X}_t, \overline{\Theta}, a_t))$$

Note the replacement of $\hat{Q}(s_t, a_t)$ in the original algorithm description with $\hat{A}(s_t, a_t)$. In order to estimate the value $\hat{V}^p(s_t)$, one possibility is to maintain another set of parameters representing the value network (which is different from the Q-network). The TD-algorithm can be used to update the parameters of the value network. However, it turns out that a single value-network is enough. This is because we can use $r_t + \gamma \hat{V}^p(s_{t+1})$ in lieu of $\hat{Q}(s_t, a_t)$. This results in an advantage function, which is the same as the TD-error:

$$\hat{A}^p(s_t, a_t) = r_t + \gamma \hat{V}^p(s_{t+1}) - \hat{V}^p(s_t)$$

In other words, we need the single value-network (cf. Figure 11.5), which serves as the critic. The above approach can also be generalized to use the $TD(\lambda)$ algorithm at any value of λ.

11.6.4 Continuous Action Spaces

The methods discussed to this point were all associated with discrete action spaces. For example, in a video game, one might have a discrete set of choices such as whether to move the cursor up, down, left, and right. However, in a robotics application, one might have continuous action spaces, in which we wish to move the robot's arm a certain distance. One possibility is to discretize the action into a set of fine-grained intervals, and use the midpoint of the interval as the representative value. One can then treat the problem as one of discrete choice. However, this is not a particularly satisfying design choice. First, the ordering among the different choices will be lost by treating inherently ordered (numerical) values as categorical values. Second, it blows up the space of possible actions, especially if the action space is multidimensional (e.g., separate dimensions for distances moved by the robot's arm and leg). Such an approach can cause overfitting, and greatly increase the amount of data required for learning.

A commonly used approach is to allow the neural network to output the parameters of a continuous distribution (e.g., mean and standard deviation of Gaussian), and then sample from the parameters of that distribution in order to compute the value of the action in the next step. Therefore, the neural network will output the mean μ and standard deviation σ for the distance moved by the robotic arm, and the actual action a will be sampled from the Gaussian $\mathcal{N}(\mu, \sigma)$ with this parameter:

$$a \sim \mathcal{N}(\mu, \sigma) \tag{11.27}$$

In this case, the action a represents the distance moved by the robot arm. The values of μ and σ can be learned using backpropagation. In some variations, σ is fixed up front as a hyper-parameter, with only the mean μ needing to be learned. The likelihood ratio trick also applies to this case, except that we use the logarithm of the density at a, rather than the discrete probability of the action a.

Advantages and Disadvantages of Policy Gradients

Policy gradient methods represent the most natural choice in applications like robotics that have continuous sequences of states and actions. For cases in which there are multidimensional and continuous action spaces, the number of possible combinations of actions can be very large. Since Q-learning methods require the computation of the maximum Q-value over all such actions, this step can turn out to be computationally intractable. Furthermore, policy gradient methods tend to be stable and have good convergence properties. However, policy gradient methods are susceptible to local minima. While Q-learning methods can sometimes oscillate around particular solutions in an unstable way, they have better capacity to reach near global optima.

11.7 Monte Carlo Tree Search

Monte Carlo tree search can be viewed as a variant of the concept of Monte Carlo rollouts in reinforcement learning. The main difference is that a tree structure is explicitly constructed showing the hierarchical relationships among the states, as the are encountered. This tree structure can also be combined with bootstrapping methods like temporal difference learning. It is also leveraged as a probabilistic alternative to the deterministic minimax trees that are used by conventional game-playing software (although the applicability is not restricted to games).

As the name implies, Monte Carlo tree search constructs a tree of states during reinforcement learning. Each node in the tree corresponds to a state, and each branch corresponds to a possible action. The tree grows over time during the search as new states are encountered. The goal of the tree search is to select the best branch to recommend the predicted action of the agent. Each branch is associated with a value based on previous outcomes in tree search from that branch as well as an upper bound "bonus" that reduces with increased exploration. This value is used to set the priority of the branches during exploration. The learned goodness of a branch is adjusted after each exploration, so that branches leading to positive outcomes are favored in later explorations.

In the following, we will describe the Monte Carlo tree search used in *AlphaGo* as a case study for exposition. Assume that the probability $P(s, a)$ of each action (move) a at state (board position) s can be estimated using a policy network. At the same time, for each move we have a quantity $Q(s, a)$, which is the quality of the move a at state s. For example, the value of $Q(s, a)$ increases with increasing number of wins by following action a from state s in simulations. The *AlphaGo* system uses a more sophisticated algorithm that also incorporates some neural evaluations of the board position after a few moves (cf. section 11.8.1). Then, in each iteration, the "upper bound" $u(s, a)$ of the quality of the move a at state s is given by the following:

$$u(s, a) = Q(s, a) + K \cdot \frac{P(s, a)\sqrt{\sum_b N(s, b)}}{N(s, a) + 1} \tag{11.28}$$

Here, $N(s, a)$ is the number of times that the action a was followed from state s over the course of the Monte Carlo tree search. In other words, the upper bound is obtained by starting with the quality $Q(s, a)$, and adding a "bonus" to it that depends on $P(s, a)$ and the number of times that branch is followed. The idea of scaling $P(s, a)$ by the number of visits is to discourage frequently visited branches and encourage greater exploration. The Monte Carlo approach is based on the strategy of selecting the branch with the largest upper bound, as in multi-armed bandit methods (cf. section 11.2). Here, the second term on the right-hand side of Equation 11.28 plays the role of providing the confidence interval for computing the upper bound. As the branch is played more and more, the exploration "bonus" for that branch is reduced, because the width of its confidence interval drops. The hyperparameter K controls the degree of exploration.

At any given state, the action a with the largest value of $u(s, a)$ is followed. This approach is applied recursively until following the optimal action does not lead to an existing node. This new state s' is now added to the tree as a leaf node with initialized values of each $N(s', a)$ and $Q(s', a)$ set to 0. Note that the simulation up to a leaf node is fully deterministic, and no randomization is involved because $P(s, a)$ and $Q(s, a)$ are deterministically computable. Monte Carlo simulations are used to estimate the value of the newly added leaf node s'. Specifically, Monte Carlo rollouts from the policy network (e.g., using $P(s, a)$ to sample actions) return either $+1$ or -1, depending on win or loss. After evaluating the leaf node, the values of $Q(s'', a'')$ and $N(s'', a'')$ on all edges (s'', a'') on the path from the current state s to the leaf s' are updated. The value of $Q(s'', a'')$ is maintained as the average value of all the evaluations at leaf nodes reached from that branch during the Monte Carlo tree search. After multiple searches have been performed from s, the most visited edge is selected as the relevant one, and is reported as the desired action.

Use in Bootstrapping

Traditionally, Monte Carlo tree search provides an improved estimate $Q(s, a)$ of the value of a state-action pair by performing repeated Monte Carlo rollouts. However, approaches that work with rollouts can often be implemented with bootstrapping instead of Monte Carlo rollouts (Intuition 11.5.1). Monte Carlo tree search provides an excellent alternative to n-step temporal-difference methods. One point about on-policy n-step temporal-difference methods is that they explore a single sequence of n-moves with the ϵ-greedy policy, and therefore tend to be too weak (with increased depth but not width of exploration). One way to strengthen them is to examine all possible n-sequences and use the optimal one with an off-policy technique (i.e., generalizing Bellman's 1-step approach). In fact, this was the approach used in Samuel's checkers program [438], which used the best option in the minimax tree for bootstrapping (and later referred to as *TD-Leaf* [23]). This results in increased complexity of exploring all possible n-sequences. Monte Carlo tree search can provide a robust alternative for bootstrapping, because it can explore multiple branches from a node to generate averaged target values. For example, the lookahead-based ground truth can use the averaged performance over all the explorations starting at a given node.

AlphaGo Zero [467] bootstraps policies rather than state values, which is extremely rare. *AlphaGo Zero* uses the relative visit probabilities of the branches at each node as *posterior* probabilities of the actions at that state. These posterior probabilities are improved over the probabilistic outputs of the policy network by virtue of the fact that the visit decisions use knowledge about the future (i.e., evaluations at deeper nodes of the Monte Carlo tree). The posterior probabilities are therefore bootstrapped as ground-truth values with respect to the policy network probabilities and used to update the weight parameters (cf. section 11.8.1).

11.8 Case Studies

In the following, we present case studies from real domains to showcase different reinforcement learning settings. We will present examples of reinforcement learning in *Go*, robotics, conversational systems, self-driving cars, and neural-network hyperparameter learning.

11.8.1 AlphaGo and AlphaZero for Go and Chess

Go is a two-person board game like chess. The complexity of a two-person board game largely depends on the size of the board and the number of valid moves at each position. The simplest example of a board game is tic-tac-toe with a 3×3 board, and most humans can solve it optimally without the need for a computer. Chess is a significantly more complex game with an 8×8 board, although clever variations of the brute-force approach of *selectively* exploring the minimax tree of moves up to a certain depth can perform significantly better than the best human today. *Go* occurs at the extreme end of complexity because of its 19×19 board.

Players play with white or black *stones*, which are kept in bowls next to the *Go* board. An example of a *Go* board is shown in Figure 11.7. The game starts with an empty board, and it fills up as players put stones on the board. Black makes the first move and starts with 181 stones in her bowl, whereas white starts with 180 stones. The total number of junctions is equal to the total number of stones in the bowls of the two players. A player places a stone of her color in each move at a particular position (from the bowl), and does not move it once it is placed. A stone of the opponent can be captured by encircling it.

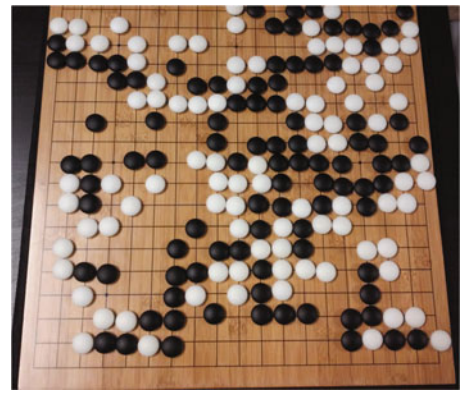


Figure 11.7: Example of a *Go* board with stones.

The objective of the game is for the player to control a larger part of the board than her opponent by encircling it with her stones.

Whereas one can make about 35 possible moves (i.e., tree branch factor) in a particular position in chess, the average number of possible moves at a particular position in *Go* is 250, which is almost an order of magnitude larger. Furthermore, the average number of sequential moves (i.e., tree depth) of a game of *Go* is about 150, which is around twice as large as chess. All these aspects make *Go* a much harder candidate from the perspective of automated game-playing. The typical strategy of chess-playing software is to construct a minimax tree with all combinations of moves the players can make up to a certain depth, and then evaluate the final board positions with chess-specific heuristics (such as the amount of remaining material and the safety of various pieces). Suboptimal parts of the tree are pruned in a heuristic manner. This approach is simply a improved version of a brute-force strategy in which all possible positions are explored up to a given depth. The number of nodes in the minimax tree of *Go* is larger than the number of atoms in the observable universe, even at modest depths of analysis (20 moves for each player). As a result of the importance of spatial intuition in these settings, humans always perform better than brute force strategies at *Go*. The use of reinforcement learning in *Go* is much closer to what humans attempt to do. We rarely try to explore all possible combinations of moves; rather, we visually learn patterns on the board that are predictive of advantageous positions, and try to make moves in directions that are expected to improve our advantage.

The automated learning of spatial patterns that are predictive of good performance is achieved with a convolutional neural network. The state of the system is encoded in the board position at a particular point, although the board representation in *AlphaGo* includes some additional features about the status of junctions or the number of moves since a stone was played. Multiple such spatial maps are required in order to provide full knowledge of the state. For example, one feature map would represent the status of each intersection, another would encode the number of turns since a stone was played, and so on. Integer feature maps were encoded into multiple one-hot planes. Altogether, the game board could be represented using 48 binary planes of 19×19 pixels.

AlphaGo uses its win-loss experience with repeated game playing (both using the moves of expert players and with games played against itself) to learn good policies for moves in various positions with a policy network. Furthermore, the evaluation of each position on the *Go* board is achieved with a value network. Subsequently, Monte Carlo tree search is

used for final inference. Therefore, *AlphaGo* is a multi-stage model, whose components are discussed in the following sections.

Policy Networks

The policy network takes as its input the aforementioned visual representation of the board, and outputs the probability of action a in state s. This output probability is denoted by $p(s, a)$. Note that the actions in the game of *Go* correspond to the probability of placing a stone at each legal position on the board. Therefore, the output layer uses the softmax activation. Two separate policy networks are trained using different approaches. The two networks were identical in structure, containing convolutional layers with ReLU nonlinearities. Each network contained 13 layers. Most of the convolutional layers convolve with 3×3 filters, except for the first and final convolutions. The first and final filters convolve with 5×5 and 1×1 filters, respectively. The convolutional layers were zero padded to maintain their size, and 192 filters were used. The ReLU nonlinearity was used, and no maxpooling was used in order to maintain the spatial footprint.

The networks were trained in the following two ways:

- *Supervised learning:* Randomly chosen samples from expert players were used as training data. The input was the state of the network, while the output was the action performed by the expert player. The score (advantage) of such a move was always $+1$, because the goal was to train the network to *imitate* expert moves, which is also referred to as *imitation learning*. Therefore, the neural network was backpropagated with the log-likelihood of the probability of the chosen move as its gain. This network is referred to as the SL-policy network. It is noteworthy that these supervised forms of imitation learning are often quite common in reinforcement learning for avoiding cold-start problems. However, subsequent work [466] showed that dispensing with this part of the learning was a better option.

- *Reinforcement learning:* In this case, reinforcement learning was used to train the network. One issue is that *Go* needs two opponents, and therefore the network was played against itself in order to generate the moves. The current network was always played against a randomly chosen network from a few iterations back, so that the reinforcement learning could have a pool of randomized opponents. The game was played until the very end, and then an advantage of $+1$ or -1 was associated with each move depending on win or loss. This data was then used to train the policy network. This network was referred to as the RL-policy network.

These networks are formidable *Go* players in their own right, and they were combined with Monte Carlo tree search to strengthen them.

Value Networks

This network was also a convolutional neural network, which uses the state of the network as the input and the predicted score in $[-1, +1]$ as output, where $+1$ indicates a perfect probability of 1. The output is the predicted score of the next player, whether it is white or black, and therefore the input also encodes the "color" of the pieces in terms of "player" or "opponent" rather than white or black. The architecture of the value network was very similar to the policy network, except that there were some differences in terms of the input and output. The input contained an additional feature corresponding to whether the next player to play was white or black. The score was computed using a single tanh unit at

the end, and therefore the value lies in the range $[-1, +1]$. The early convolutional layers of the value network are the same as those in the policy network, although an additional convolutional layer is added in layer 12. A fully connected layer with 256 units and ReLU activation follows the final convolutional layer. In order to train the network, one possibility is to use positions from a data set [630] of *Go* games. However, the preferred choice was to generate the data set using self-play with the SL-policy and RL-policy networks all the way to the end, so that the final outcomes were generated. The state-outcome pairs were used to train the convolutional neural network. Since the positions in a single game are correlated, using them sequentially in training causes overfitting. It was important to sample positions from different games in order to prevent overfitting caused by closely related training examples. Therefore, each training example was obtained from a distinct game of self-play.

Monte Carlo Tree Search

A simplified variant of Equation 11.28 was used for exploration, which is equivalent to setting $K = 1/\sqrt{\sum_b N(s, b)}$ at each node s. Section 11.7 described a version of the Monte Carlo tree search method in which only the RL-policy network is used for evaluating leaf nodes. In the case of *AlphaGo*, two approaches are combined. First, fast Monte Carlo rollouts were used from the leaf node to create evaluation e_1. While it is possible to use the policy network for rollout, *AlphaGo* trained a simplified softmax classifier with a database of human games and some hand-crafted features for faster speed of rollouts. Second, the value network created a separate evaluation e_2 of the leaf nodes. The final evaluation e is a convex combination of the two evaluations as $e = \beta e_1 + (1 - \beta)e_2$. The value of $\beta = 0.5$ provided the best performance, although using only the value network also provided closely matching performance (and a viable alternative). The most visited branch in Monte Carlo tree search was reported as the predicted move.

AlphaZero: Enhancements to Zero Human Knowledge

A later enhancement of the idea, referred to as *AlphaGo Zero* [466], removed the need for human expert moves (or an SL-network). Instead of separate policy and value networks, a single network outputs both the policy (i.e., action probabilities) $p(s, a)$ and the value $v(s)$ of the position. The cross-entropy loss on the output policy probabilities and the squared loss on the value output were added to create a single loss. Whereas the original version of *AlphaGo* used Monte Carlo tree search only for inference from trained networks, the zero-knowledge versions also use the visit counts in Monte Carlo tree search for training. One can view the visit count of each branch in tree search as a policy *improvement* operator over $p(s, a)$ by virtue of its lookahead-based exploration. This provides a basis for creating boot-strapped ground-truth values (Intuition 11.5.1) for neural network learning. While temporal difference learning bootstraps state values, this approach bootstraps visit counts for learning policies. The predicted probability of Monte Carlo tree search for action a in board state s is $\pi(s, a) \propto N(s, a)^{1/\tau}$, where τ is a temperature parameter. The value of $N(s, a)$ is computed using a similar Monte Carlo search algorithm as used for *AlphaGo*, where the *prior* probabilities $p(s, a)$ output by the neural network are used for computing Equation 11.28. The value of $Q(s, a)$ in Equation 11.28 is set to the average value output $v(s')$ from the neural network of the newly created leaf nodes s' reached from state s.

AlphaGo Zero updates the neural network by bootstrapping $\pi(s, a)$ as a ground-truth, whereas ground-truth *state values* are generated with Monte Carlo simulations. At each state

s, the probabilities $\pi(s, a)$, values $Q(s, a)$ and visit counts $N(s, a)$ are updated by running the Monte Carlo tree search procedure (repeatedly) starting at state s. The neural network from the previous iteration is used for selecting branches according to Equation 11.28 until a state is reached that does not exist in the tree or a terminal state is reached. For each non-existing state, a new leaf is added to the tree with its Q-values and visit values initialized to zero. The Q-values and visit counts of all edges on the path from s to the leaf node are updated based on leaf evaluation by the neural network (or by game rules for terminal states). After multiple searches starting from node s, the *posterior* probability $\pi(s, a)$ is used to sample an action for self-play and reach the next node s'. The entire procedure discussed in this paragraph is repeated at node s' to recursively obtain the next position s''. The game is recursively played to completion and the final value from $\{-1, +1\}$ is returned as the ground-truth value $z(s)$ of uniformly sampled states s on the game path. Note that $z(s)$ is defined from the perspective of the player at state s. The ground-truth values of the probabilities are already available in $\pi(s, a)$ for various values of a. Therefore, one can create a training instance for the neural network containing the input representation of state s, the bootstrapped ground-truth probabilities in $\pi(s, a)$, and the Monte Carlo ground-truth value $z(s)$. This training instance is used to update the neural network parameters. Therefore, if the probability and value outputs for the neural network are $p(s, a)$ and $v(s)$, respectively, the loss for a neural network with weight vector $\overline{W}$ is as follows:

$$L = [v(s) - z(s)]^2 - \sum_a \pi(s, a) \log[p(s, a)] + \lambda ||\overline{W}||^2 \tag{11.29}$$

Here, $\lambda > 0$ is the regularization parameter.

Further advancements were proposed in the form of *Alpha Zero* [467], which could play multiple games such as *Go*, shogi, and chess. *AlphaZero* has handily defeated the best chess-playing software, *Stockfish*, and has also defeated the best shogi software (*Elmo*). The victory in chess was particularly unexpected by most top players, because it was always assumed that chess required too much domain knowledge for a reinforcement learning system to win over a system with hand-crafted evaluations.

Comments on Performance

AlphaGo has shown extraordinary performance against a variety of computer and human opponents. Against a variety of computer opponents, it won the majority of the games even in handicapped settings. It also defeated the best human opponents, such as the European champion, the World champion, and the top-ranked player.

A more notable aspect of its performance was the way in which it achieved its victories. In several of its games, *AlphaGo* made many unconventional and brilliantly unorthodox moves, which would sometimes make sense only in hindsight after the victory of the program [631, 632]. There were cases in which the moves made by *AlphaGo* were contrary to conventional wisdom, but eventually revealed innovative insights acquired by *AlphaGo* during self-play. After this match, some top *Go* players reconsidered their approach to the entire game.

The performance of *Alpha Zero* in chess was similar, where it often made material sacrifices in order to incrementally improve its position and constrict its opponent. This type of behavior is a hallmark of human play and is very different from conventional chess software (which is already much better than humans). Unlike hand-crafted evaluations, it seemed to have no pre-conceived notions on the material values of pieces, or on when a king was safe in the center of the board. Furthermore, it discovered most well-known chess openings on its own using self-play, and seemed to have its own opinions on which ones

were "better." In other words, it had the ability to discover knowledge on its own. A key difference of reinforcement learning from supervised learning is that *it has the ability to innovate beyond known knowledge through learning by reward-guided trial and error*. This behavior represents some promise in other applications.

11.8.2 Self-Learning Robots

Self-learning robots represent an important frontier in artificial intelligence, in which robots can be trained to perform various tasks such as locomotion, mechanical repairs, or object retrieval by using a reward-driven approach. For example, consider the case in which one has constructed a robot that is *physically* capable of locomotion (in terms of how it is constructed and the movement choices available to it), but it has to learn the precise *choice* of movements in order to keep itself balanced and move from point A to point B. As bipedal humans, we are able to walk and keep our balance naturally without even thinking about it, but this is not a simple matter for a bipedal robot in which an incorrect choice of joint movement could easily cause it to topple over. The problem becomes even more difficult when uncertain terrain and obstacles are placed in the way of a robot.

This type of problem is naturally suited to reinforcement learning, because it is easy to judge whether a robot is walking correctly, but it is hard to specify precise rules about what the robot should do in every possible situation. In the reward-driven approach of reinforcement learning, the robot is given (virtual) rewards every time it makes progress in locomotion from point A to point B. Otherwise, the robot is free to take any actions, and it is not pre-trained with knowledge about the specific choice of actions that would help keep it balanced and walk. In other words, the robot is not seeded with any knowledge of what walking looks like (beyond the fact that it will be rewarded for using its available actions for making progress from point A to point B). This is a classical example of reinforcement learning, because the robot now needs to learn the specific sequence of actions to take in order to earn the goal-driven rewards. Although we use locomotion as a specific example in this case, this general principle applies to any type of learning in robots. For example, a second problem is that of teaching a robot manipulation tasks such as grasping an object or screwing the cap on a bottle. In the following, we will provide a brief discussion of both cases.

11.8.2.1 Deep Learning of Locomotion Skills

In this case, locomotion skills were taught to virtual robots [452], in which the robot was simulated with the *MuJoCo* physics engine [633], which stands for *Multi-Joint Dynamics with Contact*. It is a physics engine aiming to facilitate research and development in robotics, biomechanics, graphics, and animation, where fast and accurate simulation is needed without having to construct an actual robot. Both a humanoid and a quadruped robot were used. An example of the biped model is shown in Figure 11.8. The advantage of this type of simulation is that it is inexpensive to work with a virtual simulation, and one avoids the natural safety and expense issues that arise with the physical damages in an experimentation framework that is likely to be marred by high levels of mistakes/accidents. On the flip

Figure 11.8: Example of the virtual humanoid robot. Original image is available at [633].

side, a physical model provides more realistic results. In general, a simulation can often be used for smaller scale testing before building a physical model.

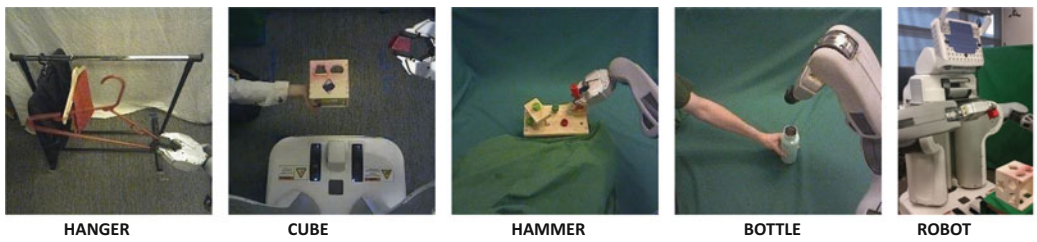
The humanoid model has 33 state dimensions and 10 actuated degrees of freedom, while the quadruped model has 29 state dimensions and 8 actuated degrees of freedom. Models were rewarded for forward progress, although episodes were terminated when the center of mass of the robot fell below a certain point. The actions of the robot were controlled by joint torques. A number of features were available to the robot, such as sensors providing the positions of obstacles, the joint positions, angles, and so on. These features were fed into the neural networks. Two neural networks were used; one was used for value estimation, and the other was used for policy estimation. Therefore, a policy gradient method was used in which the value network played the role of estimating the advantage. Such an approach is an instantiation of an actor-critic method.

A feed-forward neural network was used with three hidden layers, with 100, 50, and 25 tanh units, respectively. The approach in [452] requires the estimation of both a policy function and a value function, and the same architecture was used in both cases for the hidden layers. However, the value estimator required only one output, whereas the policy estimator required as many outputs as the number of actions. Therefore, the main difference between the two architectures was in terms of how the output layer and the loss function was designed. The generalized advantage estimator (GAE) was used in combination with trust-based policy optimization (TRPO). The bibliographic notes contain pointers to specific details of these methods. On training the neural network for 1000 iterations with reinforcement learning, the robot learned to walk with a visually pleasing gait. A video of the final results of the robot walking is available at [634]. Similar results were also later released by Google DeepMind with more extensive abilities of avoiding obstacles or other challenges [197].

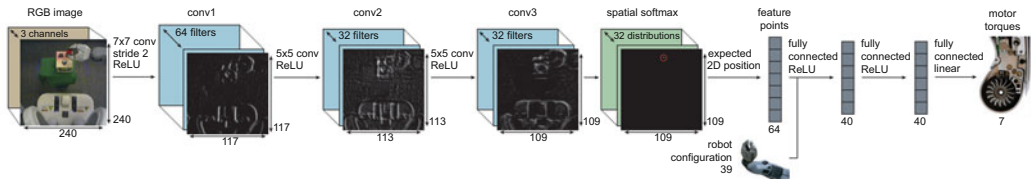
11.8.2.2 Deep Learning of Visuomotor Skills

A second and interesting case of reinforcement learning is provided in [292], in which a robot was trained for several household tasks such as placing a coat hanger on a rack, inserting a block into a shape-sorting cube, fitting the claw of a toy hammer under a nail with various grasps, and screwing a cap onto a bottle. Examples of these tasks are illustrated in Figure 11.9(a) along with an image of the robot. The actions were 7-dimensional joint motor torque commands, and each action required a sequence of commands in order to optimally perform the task. In this case, an actual physical model of a robot was used for training. A camera image was used by the robot in order to locate the objects and manipulate them. This camera image can be considered the robot's eyes, and the convolutional neural network used by the robot works on the same conceptual principle as the visual cortex (based on Hubel and Wiesel's experiments). Even though this setting seems very different from that of the Atari video games at first sight, there are significant similarities in terms of how image frames can help in mapping to policy actions. For example, the Atari setting also works with a convolutional neural network on the raw pixels. However, there were some additional inputs here, corresponding to the robot and object positions. These tasks require a high level of learning in visual perception, coordination, and contact dynamics, all of which need to learned automatically.

A natural approach is to use a convolutional neural network for mapping image frames to actions. As in the case of Atari games, spatial features need to be learned in the layers of the convolutional neural network that are suitable for earning the relevant rewards in a task-sensitive manner. The convolutional neural network had 7 layers and 92,000 parameters. The first three layers were convolutional layers, the fourth layer was a spatial softmax, and the fifth layer was a fixed transformation from spatial feature maps to a concise set of two coordinates. The idea was to apply a softmax function to the responses across the spatial feature map. This provides a probability of each position in the feature map. The expected



(a) Visuomotor tasks learned by robot



(b) Architecture of the convolutional neural network

Figure 11.9: Deep learning of visuomotor skills. These figures appear in [292]. (©2016 Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel)

position using this probability distribution provides the 2-dimensional coordinate, which is referred to as a *feature point*. Note that each spatial feature map in the convolution layer creates a feature point. The feature point can be viewed as a kind of soft argmax over the spatial probability distribution. The fifth layer was quite different from what one normally sees in a convolutional neural network, and was designed to create a precise representation of the visual scene that was suitable for feedback control. The spatial feature points are concatenated with the robot's configuration, which is an additional input occurring only after the convolution layers. This concatenated feature set is fed into two fully connected layers, each with 40 rectified units, followed by linear connections to the torques. Note that only the observations corresponding to the camera were fed to the first layer of the convolutional neural network, and the observations corresponding to the robot state were fed to the first fully connected layer. This is because the convolutional layers cannot make much use of the robot states, and it makes sense to concatenate the state-centric inputs after the visual inputs have been processed by the convolutional layers. The entire network contained about 92,000 parameters, of which 86,000 were in the convolutional layers. The architecture of the convolutional neural network is shown in Figure 11.9(b). The observations consist of the RGB camera image, joint encoder readings, velocities, and end-effector pose.

The full robot states contained between 14 and 32 dimensions, such as the joint angles, end-effector pose, object positions, and their velocities. This provided a practical notion of a state. As in all policy-based methods, the outputs correspond to the various actions (motor torques). One interesting aspect of the approach discussed in [292] is that it transforms the reinforcement learning problem into supervised learning. A *guided policy search* method was used, which is not discussed in this chapter. This approach converts portions of the reinforcement learning problem into supervised learning. Interested readers are referred to [292], where a video of the performance of the robot (trained using this system) may also be found.

11.8.3 Building Conversational Systems: Deep Learning for Chatbots

Chatbots are also referred to as *conversational systems* or *dialog systems*. The ultimate goal of a chatbot is to build an agent that can freely converse with a human about a variety of topics in a natural way. We are very far from achieving this goal. However, significant progress has been made in building chatbots for specific domains and particular applications (e.g., negotiation or shopping assistant). An example of a relatively general-purpose system is Apple's Siri, which is a digital personal assistant. One can view Siri as an open-domain system, because it is possible to have conversations with it about a wide variety of topics. It is reasonably clear to anyone using Siri that the assistant is sometimes either unable to provide satisfactory responses to difficult questions, and in some cases hilarious responses to common questions are hard-coded. This is, of course, natural because the system is relatively general-purpose, and we are nowhere close to building a human-level conversational system. In contrast, closed-domain systems have a specific task in mind, and can therefore be more easily trained in a reliable way.

In the following, we will describe a system built by *Facebook* for end-to-end learning of negotiation skills [296]. This is a closed-domain system because it is designed for the

particular purpose of negotiation. As a test-bed, the following negotiation task was used. Two agents are shown a collection of items of different types (e.g., two books, one hat, three balls). The agents are instructed to divide these items among themselves by negotiating a split of the items. A key point is that the value of each of the types of items is different for the two agents, but they are not aware of the value of the items for each other. This is often the case in real-life negotiations, where users attempt to reach a mutually satisfactory outcome by negotiating for items of value to them.

The values of the items are always assumed to be non-negative and generated randomly in the test-bed under some constraints. First, the total value of all items for a user is 10. Second, each item has non-zero value to at least one user so that it makes little sense to ignore an item. Last, some items have nonzero values to both users. Because of these constraints, it is impossible for both users to achieve the maximum score of 10, which ensures a competitive negotiation process. After 10 turns, the agents are allowed the option to complete the negotiation with no agreement, which has a value of 0 points for both users. The three item types of books, hats, and balls were used, and a total of between 5 and 7 items existed in the pool. The fact that the values of the items are different for the two users (without knowledge about each other's assigned values) is significant; if both negotiators are capable, they will be able to achieve a total value of larger than 10 for the items between them. Nevertheless, the better negotiator will be able to capture the larger value by optimally negotiating for items with a high value for them.

The reward function for this reinforcement learning setting is the final value of the items attained by the user. One can use supervised learning on previous dialogs in order to maximize the likelihood of utterances. A straightforward use of recurrent networks to maximize the likelihood of utterances resulted in agents that were too eager to compromise. Therefore, the approach combined supervised learning with reinforcement learning. The incorporation of supervised learning within the reinforcement learning helps in ensuring that the models do not diverge from human language. A form of planning for dialogs called *dialog roll-out* was introduced. The approach uses an encoder-decoder recurrent architecture, in which the decoder maximizes the reward function rather than the likelihood of utterances. This encoder-decoder architecture is based on sequence-to-sequence learning, as discussed in section 8.7.3 of Chapter 8.

To facilitate supervised learning, dialogs were collected from *Amazon Mechanical Turk*. A total of 5808 dialogs were collected in 2236 unique scenarios, where a scenario is defined by assignment of a particular set of values to the items. Of these cases, 252 scenarios corresponding to 526 dialogs were held out. Each scenario results in two training examples, which are derived from the perspective of each agent. A concrete training example could be one in which the items to be divided among the two agents correspond to 3 books, 2 hats, and 1 ball. These are part of the input to each agent. The second input could be the value of each item to the agent, which are (i) Agent A: book:1, hat:3, ball:1, and (ii) Agent B: book:2, hat:1, ball:2. Note that this means that agent A should secretly try to get as many hats as possible in the negotiation, whereas agent B should focus on books and balls. An example of a dialog in the training data is given below [296]:

Agent A: I want the books and the hats, you get the ball.
Agent B: Give me a book too and we have a deal.
Agent A: Ok, deal.
Agent B: ⟨choose⟩

The final output for agent A is 2 books and 2 hats, whereas the final output for agent B is 1 book and 1 ball. Therefore, each agent has her own set of inputs and outputs, and the

dialogs for each agent are also viewed from their own perspective in terms of the portions that are reads and the portions that are writes. Therefore, each scenario generates two training examples and the same recurrent network is shared for generating the writes and the final output of each agent. The dialog x is a list of tokens $x_0 \ldots x_T$, containing the turns of each agent interleaved with symbols marking whether the turn was written by an agent or their partner. A special token at the end indicates that one agent has marked that an agreement has been reached.

The supervised learning procedure uses four different gated recurrent units (GRUs). The first gated recurrent unit GRU_g encodes the input goals, the second gated recurrent unit GRU_q generates the terms in the dialog, a forward-output gated recurrent unit $GRU_{\overrightarrow{o}}$, and a backward-output gated recurrent unit $GRU_{\overleftarrow{o}}$. The output is essentially produced by a bi-directional GRU. These GRUs are hooked up in end-to-end fashion. In the supervised learning approach, the parameters are trained using the inputs, dialogs, and outputs available from the training data. The loss for the supervised model for a weighted sum of the token-prediction loss of the dialog and the output choice prediction loss of the items.

However, for reinforcement learning, dialog roll-outs are used. Note that the group of GRUs in the supervised model is, in essence, providing probabilistic outputs. Therefore, one can adapt the same model to work for reinforcement learning by simply changing the loss function. In other words, the GRU combination can be considered a type of policy network. One can use this policy network to generate Monte Carlo roll-outs of various dialogs and their final rewards. Each of the sampled actions becomes a part of the training data, and the action is associated with the final reward of the roll-out. In other words, the approach uses *self-play* in which the agent negotiates with itself to learn better strategies. The final reward achieved by a roll-out is used to update the policy network parameters. This reward is computed based on the value of the items negotiated at the end of the dialog. This approach can be viewed as an instance of the REINFORCE algorithm [552]. One issue with self-play is that the agents tend to learn their own language, which deviates from natural human language when both sides use reinforcement learning. Therefore, one of the agents is constrained to be a supervised model.

For the final prediction, one possibility is to directly sample from the probabilities output by the GRU. However, such an approach is often not optimal when working with recurrent networks. Therefore, a two-stage approach is used. First, c candidate utterances are created by using sampling. The expected reward of each candidate utterance is computed and the one with the largest expected value is selected. In order to compute the expected reward, the output was scaled by the probability of the dialog because low-probability dialogs were unlikely to be selected by either agent.

A number of interesting observations were made in [296] about the performance of the approach. First, the supervised learning methods often tended to give up easily, whereas the reinforcement learning methods were more persistent in attempting to obtain a good deal. Second, the reinforcement learning method would often exhibit human-like negotiation tactics. In some cases, it feigned interest in an item that was not really of much value in order to obtain a better deal for another item.

11.8.4 Self-Driving Cars

As in the case of the robot locomotion task, the car is rewarded for progressing from point A to point B without causing accidents or other undesirable road incidents. The car is equipped with various types of video, audio, proximity, and motion sensors in order to record observations. The objective of the reinforcement learning system is for the car to go from point A to point B safely irrespective of road conditions.

Driving is a task for which it is hard to specify the proper rules of action in every situation; on the other hand, it is relatively easy to judge when one is driving correctly. This is precisely the setting that is well suited to reinforcement learning. Although a fully self-driving car would have a vast array of components corresponding to inputs and sensors of various types, we focus on a simplified setting in which a single camera is used [45, 46]. This system is instructive because it shows that even a single front-facing camera is sufficient to accomplish quite a lot when paired with reinforcement learning. Interestingly, this work was inspired by the 1989 work of Pomerleau [395], who built the *Autonomous Land Vehicle in a Neural Network (ALVINN)* system, and the main difference from the work done over 25 years back was one of increased data and computational power. In addition, the work uses some advances in convolutional neural networks for modeling. Therefore, this work showcases the great importance of increased data and computational power in building reinforcement learning systems.

The training data was collected by driving in a wide variety of roads and conditions. The data was collected primarily from central New Jersey, although highway data was also collected from Illinois, Michigan, Pennsylvania, and New York. Although a single front-facing camera in the driver position was used as the primary data source for making decisions, the training phase used two additional cameras at other positions in the front to collect rotated and shifted images. These auxiliary cameras, which were not used for final decision making, were however useful for collecting additional data. The placement of the additional cameras ensured that their images were shifted and rotated, and therefore they could be used to train the network to recognize cases where the car position had been compromised. In short, these cameras were useful for data augmentation. The neural network was trained to minimize the error between the steering command output by the network and the command output by the human driver. Note that this approach tends to make the approach closer to supervised learning rather than reinforcement learning. These types of learning methods are also referred to as *imitation learning* [445]. Imitation learning is often used as a first step to buffer the cold-start inherent in reinforcement learning systems.

Scenarios involving imitation learning are often similar to those involving reinforcement learning. It is relatively easy to use reinforcement setting in this scenario by giving a reward when the car makes progress without human intervention. On the other hand, if the car either does not make progress or requires human intervention, it is penalized. However, this does not seem to be the way in which the self-driving system of [45, 46] is trained. One issue with settings like self-driving cars is that one always has to account for safety issues during training. Although published details on most of the available self-driving cars are limited, it seems that supervised learning has been the method of choice compared to reinforcement learning in this setting. Nevertheless, the differences between using supervised learning and reinforcement learning are not significant in terms of the broader architecture of the neural network that would be useful. A general discussion of reinforcement learning in the context of self-driving cars may be found in [635].

The convolutional neural network architecture is shown in Figure 11.10. The network consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The first convolutional layer used a 5×5 filter with a stride of 2. The next two convolutional layers each used non-strided convolution with a 3×3 filter. These convolutional layers were followed with three fully connected layers. The final output value was a control value, corresponding to the inverse turning radius. The network had 27 million connections and $250,000$ parameters. Specific details of how the deep neural network performs the steering are provided in [46].

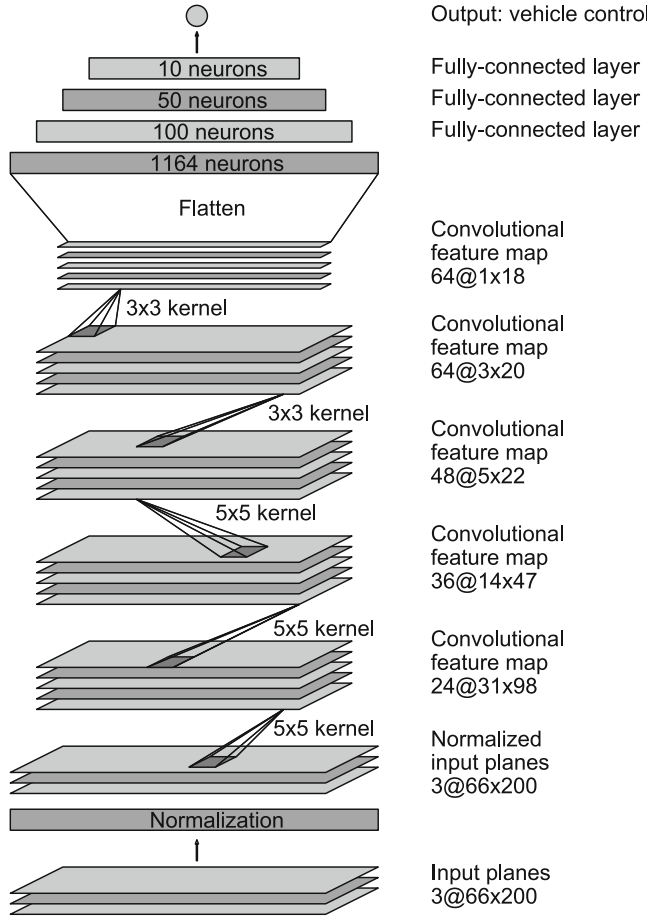


Figure 11.10: The neural network architecture of the control system in the self-driving car discussed in [45] (Courtesy NVIDIA).

The resulting car was tested both in simulation and in actual road conditions. A human driver was always present in the road tests to perform interventions when necessary. On this basis, a measure was computed on the percentage of time that human intervention was required. It was found that the vehicle was autonomous 98% of the time. Some interesting observations were obtained by visualizing the activation maps of the trained convolutional neural network (based on the methodology discussed in Chapter 9). In particular, it was observed that the features were heavily biased towards learning aspects of the image that were important to driving. In the case of unpaved roads, the feature activation maps were able to detect the outlines of the roads. On the other hand, if the car was located in a forest, the feature activation maps were full of noise. Note that this does not happen in a convolutional neural network that is trained on *ImageNet* because the feature activation maps would typically contain useful characteristics of trees, leaves, and so on. This difference in the two cases is because the convolutional network of the self-driving setting is trained in a goal-driven matter, and it learns to detect features that are relevant to driving. The specific characteristics of the trees in a forest are not relevant to driving.

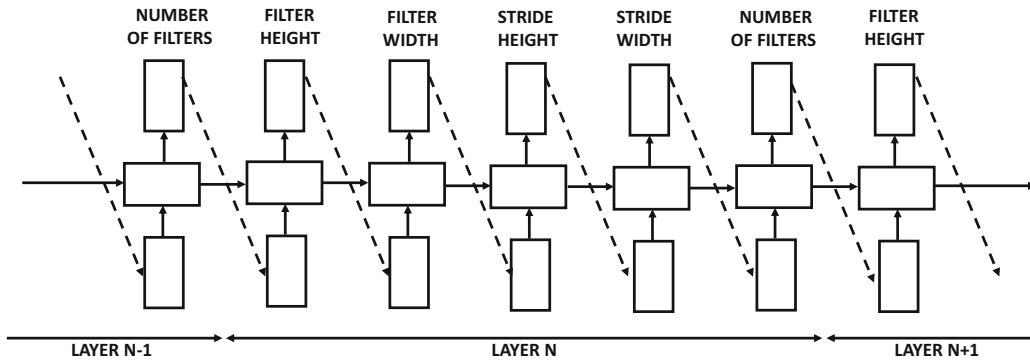


Figure 11.11: The controller network for learning the convolutional architecture of the child network [595]. The controller network is trained with the REINFORCE algorithm.

11.8.5 Neural Architecture Search with Reinforcement Learning

An interesting application of reinforcement learning is to learn the neural network architecture for performing a specific task. For discussion purposes, let us consider a setting in which we wish to determine the structure of a convolutional neural architecture for classify a data set like CIFAR-10 [607]. Clearly, the structure of the neural network depends on a number of hyper-parameters, such as the number of filters, filter height, filter width, stride height, and stride width. These parameters depend on one another, and the parameters of later layers depend on those from earlier layers.

The reinforcement learning method uses a recurrent network as the *controller* to decide the parameters of the convolutional network, which is also referred to as the *child network* [595]. The overall architecture of the recurrent network is illustrated in Figure 11.11. The choice of a recurrent network is motivated by the sequential dependence between different architectural parameters. The softmax classifier is used to predict each output as a token rather than a numerical value. This token is then used as an input into the next layer, which is shown by the dashed lines in Figure 11.11. The generation of the parameter as a token results in a discrete action space, which is generally more common in reinforcement learning as compared to a continuous action space.

The performance of the child network on a validation set drawn from CIFAR-10 is used to generate the reward signal. Note that the child network needs to be trained on the CIFAR-10 data set in order to test its accuracy. Therefore, this process requires a full training procedure of the child network, which is quite expensive. This reward signal is used in conjunction with the REINFORCE algorithm in order to train the parameters of the controller network. Therefore, the controller network is really the policy network in this case, which generates a sequence of inter-dependent parameters.

A key point is about the number of layers of the child network (which also decides the number of layers in the recurrent network). This value is not held constant but it follows a certain schedule as training progresses. In the early iterations, the number of layers is fewer, and therefore the learned architecture of the convolutional network is shallow. As training progresses, the number of layers slowly increases over time. The policy gradient method is not very different from what is discussed earlier in this chapter, except that a recurrent network is trained with the reward signal rather than a feed-forward network. Various types of optimizations are also discussed in [595], such as efficient implementations with parallelism and the learning of advanced architectural designs like skip connections.

11.9 Practical Challenges Associated with Safety

Simplifying the design of highly complex learning algorithms with reinforcement learning can sometimes have unexpected effects. By virtue of the fact that reinforcement learning systems have larger levels of freedom than other learning systems, it naturally leads to some safety related concerns. While biological greed is a powerful factor in human intelligence, it is also a source of many undesirable aspects of human behavior. The simplicity that is the greatest strength of reward-driven learning is also its greatest pitfall in biological systems. Simulating such systems therefore results in similar pitfalls from the perspective of artificial intelligence. For example, poorly designed rewards can lead to unforeseen consequences, because of the exploratory way in which the system learns its actions. Reinforcement learning systems can frequently learn unknown "cheats" and "hacks" in imperfectly designed video games, which tells us a cautionary tale of what might happen in a less-than-perfect real world. Robots learn that simply pretending to screw caps on bottles can earn faster rewards, as long as the human or automated evaluator is fooled by the action. In other words, the design of the reward function is sometimes not a simple matter.

Furthermore, a system might try to earn virtual rewards in an "unethical" way. For example, a cleaning robot might try to earn rewards by first creating messes and then cleaning them [11]. One can imagine even darker scenarios for robot nurses. Interestingly, these types of behaviors are sometimes also exhibited by humans. These undesirable similarities are a direct result of simplifying the learning process in machines by leveraging the simple greed-centric principles with which biological organisms learn. Striving for simplicity results in ceding more control to the machine, which can have unexpected effects. In some cases, there are ethical dilemmas in even designing the reward function. For example, if it becomes inevitable that an accident is going to occur, should a self-driving car save its driver or two pedestrians? Most humans would save themselves in this setting as a matter of reflexive biological instinct; however, it is an entirely different matter to incentivize a learning system to do so. At the same time, it would be hard to convince a human operator to trust a vehicle where her safety is not the first priority for the learning system. Reinforcement learning systems are also susceptible to the ways in which their human operators interact with them and manipulate the effects of their underlying reward function; there have been occasions where a chatbot was taught to make offensive or racist remarks.

Learning systems have a harder time in generalizing their experiences to new situations. This problem is referred to as *distributional shift*. For example, a self-driving car trained in one country might perform poorly in another. Similarly, the exploratory actions in reinforcement learning can sometimes be dangerous. Imagine a robot trying to solder wires in an electronic device, where the wires are surrounded with fragile electronic components. Trying exploratory actions in this setting is fraught with perils. These issues tell us that we cannot build AI systems with no regard to safety. Indeed, some organizations like *OpenAI* [636] have taken the lead in these matters of ensuring safety. Some of these issues are also discussed in [11] with broader frameworks of possible solutions. In many cases, it seems that the human would have to be involved in the loop to some extent in order to ensure safety [441].

11.10 Summary

This chapter studies the problem of reinforcement learning in which agents interact with the environment in a reward-driven manner in order to learn the optimal actions. There

are several classes of reinforcement learning methods, of which Monte Carlo methods, bootstrapping, and policy-driven methods are the most common. Many of these methods are end-to-end systems that integrate deep neural networks to take in sensory inputs and learn policies that optimize rewards. Reinforcement learning algorithms are used in many settings like playing video or other types of games, robotics, and self-driving cars. The ability of these algorithms to learn via experimentation often leads to innovative solutions that are not possible with other forms of learning. Reinforcement learning algorithms also pose unique challenges associated with safety because of the oversimplification of the learning process with reward functions.

11.11 Bibliographic Notes and Software Resources

An excellent overview on reinforcement learning may be found in the book by Sutton and Barto [499]. A number of surveys on reinforcement learning are available at [301]. David Silver's lectures on reinforcement learning are freely available on *YouTube* [641]. The method of temporal differences was proposed by Samuel in the context of a checkers program [438] and formalized by Sutton [498]. Q-learning was proposed by Watkins in [539], and a convergence proof is provided in [540]. The SARSA algorithm was introduced in [428]. Early methods for using neural networks in reinforcement learning were proposed in [305, 360, 509–511]. The work in [509] developed TD-Gammon, which was a backgammon playing program.

A system that used a convolutional neural network to create a deep Q-learning algorithm with raw pixels was pioneered in [346, 347]. It has been suggested in [346] that the approach presented in the paper can be improved with other well-known ideas such as prioritized sweeping [354]. Asynchronous methods that use multiple agents in order to perform the learning are discussed in [348].

One drawback of Q-learning is that it is known to overestimate the values of actions under certain circumstances. This problem was addressed by *double Q-learning*, which was proposed in [184]. The use of prioritized experience replay to improve the performance of reinforcement learning algorithms under sparse data is discussed in [446]. Such an approach significantly improves the performance on Atari games play.

In recent years, policy gradients have become more popular than Q-learning methods. An interesting and simplified description of this approach for the Atari game of *Pong* is provided in [629]. Early methods for using finite difference methods for policy gradients are discussed in [149, 366]. Likelihood methods for policy gradients were pioneered by the REINFORCE algorithm [552]. A number of analytical results on this class of algorithms are provided in [500]. Policy gradients have been used in for learning in the game of *Go* [465], although the overall approach combines a number of different elements. Natural policy gradients were proposed in [240]. One such method [451] has been shown to perform well at learning locomotion in robots. The use of *generalized advantage estimation (GAE)* with continuous rewards is discussed in [452]. The approach in [451, 452] uses natural policy gradients for optimization, and the approach is referred to as *trust region policy optimization* (TRPO). The basic idea is that bad gradient-descent steps have more severe consequences in reinforcement learning (as compared to supervised learning) because the quality of the collected data worsens. Therefore, the TRPO method prefers second-order methods with conjugate gradients (see Chapter 2), in which the updates tend to stay within good regions of trust. Surveys are also available on specific types of reinforcement learning methods like actor-critic methods [169].

Monte Carlo tree search was proposed in [256]. Subsequently, it was used in the game of *Go* [141, 357, 465, 466]. A survey on these methods may be found in [52]. Later versions of *AlphaGo* dispensed with the supervised portions of learning, adapted to chess and shogi, and performed better with zero initial knowledge [466, 467]. The *AlphaGo* approach combines several ideas, including the use of policy networks, Monte Carlo tree search, and convolutional neural networks. The use of convolutional neural networks for playing the game of *Go* has been explored in [76, 318, 497]. Many of these methods use supervised learning in order to mimic human experts at *Go*. Some TD-learning methods for chess, such as *NeuroChess* [513], *KnightCap* [23], and *Giraffe* [265] have been explored, but were not as successful as conventional engines. The pairing of convolutional neural networks and reinforcement learning for spatial games seems to be a new (and successful) recipe that distinguishes *AlphaZero* from these methods. Several methods for training self-learning robots are presented in [292, 451, 452]. An overview of deep reinforcement learning methods for dialog generation is provided in [298]. Conversation models that use only supervised learning with recurrent networks are discussed in [461, 528]. The negotiation chatbot discussed in this chapter is described in [296]. The description of self-driving cars is based on [45, 46]. An MIT course on self-driving cars is available at [635]. Reinforcement learning has also been used to generate structured queries from natural language [588], or for learning neural architectures in various tasks [20, 595].

Reinforcement learning can also improve deep learning models. This is achieved with the notion of *attention* [349, 562]. The idea is that large parts of the data are often irrelevant for learning, and learning how to focus on selective portions of the data can significantly improve results. The selection of relevant portions of the data is achieved with reinforcement learning. Attention mechanisms are discussed in section 12.2 of Chapter 12.

Software Resources and Testbeds

Although significant progress has been made in designing reinforcement learning algorithms in recent years, commercial software using these methods is still relatively limited. Nevertheless, numerous software testbeds are available that can be used in order to test various algorithms. Perhaps the best source for high-quality reinforcement learning baselines is available from *OpenAI* [645]. *TensorFlow* [646] and *Keras* [647] implementations of reinforcement learning algorithms are also available.

Most frameworks for testing and development of reinforcement learning algorithms are specialized to specific types of reinforcement learning scenarios. Some frameworks are lightweight, and can be used for quick testing. For example, the ELF framework [515], created by *Facebook*, is designed for real-time strategy games, and is an open-source and light-weight reinforcement learning framework. The *OpenAI Gym* [642] provides environments for development of reinforcement learning algorithms for Atari games and simulated robots. The *OpenAI Universe* [643] can be used to turn reinforcement learning programs into Gym environments. For example, self-driving car simulations have been added to this environment. An Arcade learning environment for developing agents in the context of Atari games is described in [26]. The *MuJoCo* simulator [633], which stands for Multi-Joint dynamics with Contact, is a physics engine, and is designed for robotics simulations. An application with the use of *MuJoCo* is described in this chapter. *ParlAI* [644] is an open-source framework for dialog research by *Facebook*, and is implemented in Python. Baidu has created an open-source platform of its self-driving car project, referred to as *Apollo* [648].

11.12 Exercises

1. The chapter gives a proof of the likelihood ratio trick (cf. Equation 11.25) for the case in which the action a is discrete. Generalize this result to continuous-valued actions.

2. Throughout this chapter, a neural network, referred to as the policy network, has been used in order to implement the policy gradient. Discuss the importance of the choice of network architecture in different settings.

3. You have two slot machines, each of which has an array of 100 lights. The probability distribution of the reward from playing each machine is an unknown (and possibly machine-specific) function of the pattern of lights that are currently lit up. Playing a slot machine changes its light pattern in some well-defined but unknown way. Discuss why this problem is more difficult than the multi-armed bandit problem. Design a deep learning solution to optimally choose machines in each trial that will maximize the average reward per trial at steady-state.

4. Consider the well-known game of rock-paper-scissors. Human players often try to use the history of previous moves to guess the next move. Would you use a Q-learning or a policy-based method to learn to play this game? Why? Now consider a situation in which a human player samples one of the three moves with a probability that is an unknown function of the history of 10 previous moves of each side. Propose a deep learning method that is designed to play with such an opponent. Would a well-designed deep learning method have an advantage over this human player? What policy should a human player use to ensure probabilistic parity with a deep learning opponent?

5. Consider the game of tic-tac-toe in which a reward drawn from $\{-1, 0, +1\}$ is given at the end of the game. Suppose you learn the values of all states (assuming optimal play from both sides). Discuss why states in non-terminal positions will have non-zero values. What does this tell you about credit-assignment of intermediate moves to the reward value received at the end?

6. Write a Q-learning implementation that learns the value of each state-action pair for a game of tic-tac-toe by repeatedly playing against human opponents. No function approximators are used and therefore the entire table of state-action pairs is learned using Equation 11.4. Assume that you can initialize each Q-value to 0 in the table.

7. The two-step TD-error is defined as follows:

$$\delta_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$$

   (a) Propose a TD-learning algorithm for the 2-step case.

   (b) Propose an on-policy n-step learning algorithm like SARSA. Relate this approach to other algorithms when $n = \infty$.

   (c) Propose an off-policy n-step learning algorithm like Q-learning and discuss its advantages/disadvantages with respect to (b).

8. Suppose that someone gave you a blackbox Application Programming Interface (API) that efficiently evaluated a chess position quite well using a domain-specific evaluation function (say, an extremely fast version of *Stockfish* evaluation). How can you leverage on this blackbox to generate an even stronger evaluation blackbox that is encoded within the parameters of a neural network?

9. Suppose that you play a game with an infinite sequence of unbiased coin flips in which you call either heads or tails at each flip. Your payoff at each flip outcome is a value that is an unknown hash function of the bitstring of length $2k$ containing the alternating sequence of last k calls and flip outcomes. How can you use Q-learning and/or Monte Carlo rollouts (with the tabular method) to learn how to make calls for manageable values of k. Define all states, actions, and data structures used.

10. Comment on the effectiveness of model generalization for large values of k in Exercise 9. How does it depend on the nature of the hash function?

Chapter 12

Advanced Topics in Deep Learning

> "One of the greatest gifts you can give to anyone is the gift of attention." – Jim Rohn

12.1 Introduction

This chapter covers several advanced topics in deep learning, which either do not naturally fit within the focus of the previous chapters, or because their level of complexity requires separate treatment. The topics discussed in this chapter include the following:

1. *Attention models:* Humans do not actively use all the information available to them from the environment at any given time. Rather, they focus on specific portions of the data that are relevant to the task at hand. This biological notion is referred to as that of *attention*. Similarly, neural networks with attention focus on smaller portions of the data that are relevant to the task at hand.

2. *Models with selective access to internal memory:* These models use *addressing mechanisms* to control reads and writes to the internal memory of the neural network, which results in a logical control flow that is similar to that of the programming style on a conventional computer. Such an architecture is referred to as a *memory network* or *neural Turing machine*.

3. *Generative adversarial networks:* Generative adversarial networks are generative models of data, which create realistic samples from training data by using two adversarial networks. The *generator* network creates synthetic samples, and the *discriminator* network is a classifier that distinguishes between real and synthetic samples. The two networks are simultaneously trained as adversaries, until the discriminator is no longer able to distinguish between real and synthetic samples.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0_12

In addition, this chapter visits the classical topic of *competitive learning*, a specific example of which is the *Kohonen self-organizing map*. This approach is useful for unsupervised learning applications like clustering, dimensionality reduction and compression.

Chapter Organization

This chapter is organized as follows. The next section discusses attention mechanisms in deep learning. Neural Turing machines are introduced in section 12.3. Adversarial learning is presented in section 12.4. Generative adversarial networks are discussed in section 12.5. Competitive learning methods are discussed in section 12.6. The limitations of neural networks are presented in section 12.7. A summary is presented in section 12.8.

12.2 Attention Mechanisms

Human beings rarely use all the available sensory inputs in order to accomplish specific tasks. Consider the problem of finding an address defined by a specific house number on a street. A key part of this process is to identify the specific place on the front of the house where the number is written. Although the retina often has an image of a broader house front, although one rarely focuses on the full image. The retina has a small portion, referred to as the *macula* with a central *fovea*, which has an extremely high resolution compared to the remainder of the eye. This region has a high concentration of color-sensitive cones, whereas most of the non-central portions of the eye have relatively low resolution with a predominance of color-insensitive rods. The different regions of the eye are shown in Figure 12.1. When reading a street number, the fovea *fixates* on the number, and its image falls on a portion of the retina that corresponds to the macula (and especially the fovea). Although one is aware of the other objects outside this central field of vision, it is virtually impossible to use images in the peripheral region to perform detail-oriented tasks. For example, it is very difficult to read letters projected on peripheral portions of the retina. The foveal region is a tiny fraction of the full retina, and it has a diameter of only 1.5 millimeters. The eye effectively transmits a high-resolution version of less than 0.5% of the surface area of the image that falls on the full retina. This approach is biologically advantageous, because only a carefully selected part of the image is transmitted in high resolution, and it reduces the internal processing required *for the specific task at hand*. The notion of attention selectivity is not restricted to visual aspects, but it also applies to other sensory inputs, such as hearing or smell. A similar observation applies to neural networks in which attention is applied to diverse domains such as computer vision and natural language processing.

An interesting application of attention is motivated by *Google Streetview*, which enables Web-based retrieval of street images. This type of retrieval requires the association of house images with their street numbers. Often, this information needs to be distilled from the images. Given an image of the frontal part of a house, how does one identify its street number? The main challenge here is to be able to focus on (i.e., *attend to*) the relevant portion of the image containing the street number. Therefore, an iterative approach is required in searching specific parts of the image with the use of knowledge gained from previous iterations. Here, it is useful to draw inspirations from how biological organisms work. Biological organisms draw quick visual cues from whatever they are focusing on in order to identify *where to next look* to get what they want. For example, if we first focus on the door knob by chance, then we know from experience (i.e., our trained neurons tell us) to look to its upper left or right to find the street number. This type of iterative process sounds

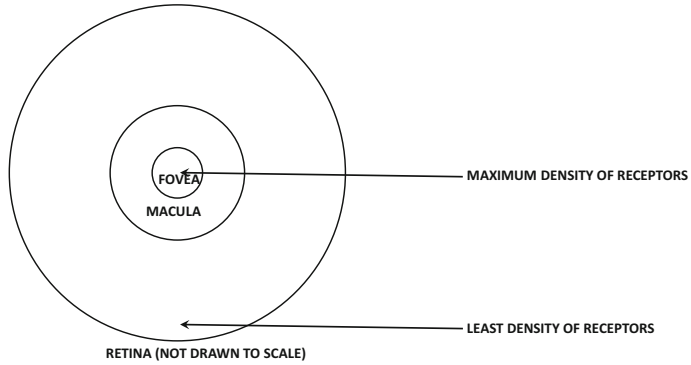


Figure 12.1: Resolutions in different retinal regions: the fovea is the center of visual attention.

a lot like the reinforcement learning methods discussed in the previous chapter, where one iteratively obtains cues from previous steps in order to learn what to do to earn *rewards* (i.e., accomplish a task like finding the street number). Although many attention mechanisms are indeed based on reinforcement learning, most modern attention mechanisms (e.g., channel weighting of SENet in section 9.4.6) learn soft weights to emphasize informative features without the use of reinforcement learning.

The notion of attention is also well suited to natural language processing in which attention is used to focus on relevant parts of the text containing an answer to a question or to focus on the appropriate portions of the source sentence that translate to specific portions of a target sentence. Most attention mechanisms in natural language processing do not use reinforcement learning, but they use soft weighting instead (which is the dominant approach to attention today). The idea of soft attention is closely related to that of supervised nearest neighbor classification in which the weights of data objects are learned in a data-driven way [397]. In the following, we will discuss various attention mechanisms for image, natural language, and graph domains.

12.2.1 Recurrent Models of Visual Attention

The work on recurrent models of visual attention [349] uses reinforcement learning to focus on important parts of an image. The idea is to use a (relatively simple) neural network in which only the resolution of specific portions of the image centered at a particular location is high. This location can change with time, as one learns more about the relevant portions of the image to explore over the course of time. Selecting a particular location in a given time-stamp is referred to as a *glimpse*. A recurrent neural network is used as the controller to identify the precise location in each time-stamp; this choice is based on the feedback from the glimpse in the previous time-stamp. The work in [349] shows that using a simple neural network (called a "glimpse network") to process the image together with the reinforcement-based training can outperform a convolutional neural network for classification.

We consider a dynamic setting in which the image may be partially observable, and the portions that are observable might vary with time-stamp t. Therefore, this setting is quite general, although we can obviously use it for simpler settings in which the image $\overline{X}_t$ is fixed in time. The overall architecture can be described in a modular way by treating specific parts of the neural network as black-boxes. These modular portions are described below:

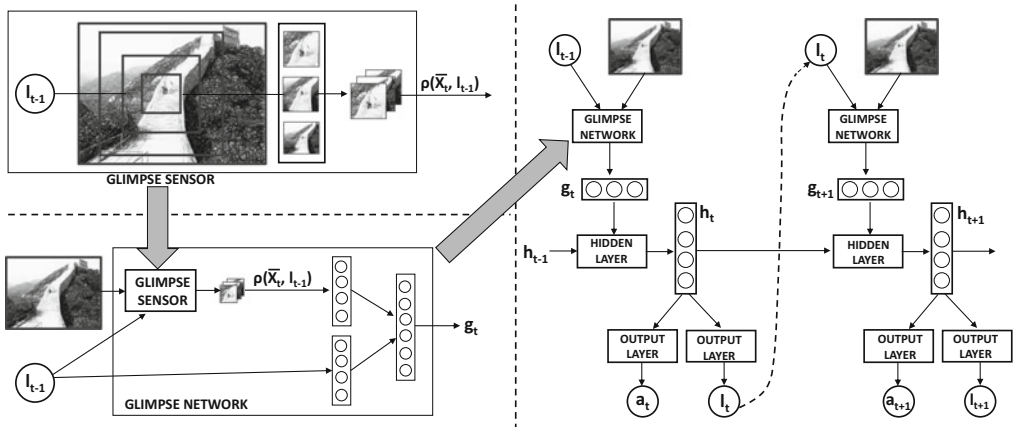


Figure 12.2: The recurrent architecture for leveraging visual attention

1. *Glimpse sensor:* Given an image with representation $\overline{X}_t$, a *glimpse sensor* creates a retina-like representation of the image. The glimpse sensor is conceptually assumed to not have full access to the image (because of bandwidth constraints), and is able to access only a small portion of the image in high-resolution, which is centered at l_{t-1}. This is similar to how the eye accesses an image in real life. The resolution of a particular location in the image reduces with distance from the location l_{t-1}. The reduced representation of the image is denoted by $\rho(\overline{X}_t, l_{t-1})$. The glimpse sensor, which is shown in the upper left corner of Figure 12.2, is a part of a larger glimpse network. This network is discussed below.

2. *Glimpse network:* The glimpse network contains the glimpse sensor and encodes both the glimpse location l_{t-1} and the glimpse representation $\rho(\overline{X}_t, l_{t-1})$ into hidden spaces using linear layers. Subsequently, the two are combined into a single hidden representation using another linear layer. The resulting output g_t is the input into the tth time-stamp of the hidden layer in the recurrent neural network. The glimpse network is shown in the lower-right corner of Figure 12.2.

3. *Recurrent neural network:* The recurrent neural network is the main network that is creating the action-driven outputs in each time-stamp (for earning rewards). The recurrent neural network includes the glimpse network, and therefore it includes the glimpse sensor as well (since the glimpse sensor is a part of the glimpse network). This output action of the network at time-stamp t is denoted by a_t, and rewards are associated with the action. In the simplest case, the reward might be the class label of the object or a numerical digit in the *Google Streetview* example. It also outputs a location l_t in the image for the next time-stamp, on which the glimpse network should focus. The output $\pi(a_t)$ is implemented as a probability of action a_t. This probability is implemented with the softmax function, as is common in policy networks (cf. Figure 11.6 of Chapter 11). The training of the recurrent network is done using the objective function of the REINFORCE framework to maximize the expected reward over time. The gain for each action is obtained by multiplying $\log(\pi(a_t))$ with the advantage of that action (cf. section 11.6.2 of Chapter 11). Therefore, the overall approach is a reinforcement learning method in which the attention locations and actionable outputs are learned simultaneously. It is noteworthy that the history of

actions of this recurrent network is encoded within the hidden states h_t. The overall architecture of the neural network is illustrated on the right-hand side of Figure 12.2. Note that the glimpse network is included as a part of this overall architecture, because the recurrent network utilizes a glimpse of the image (or current state of scene) in order to perform the computations in each time-stamp.

Note that the use of a recurrent neural network architecture is useful but not necessary in these contexts.

Reinforcement Learning

This approach is couched within the framework of reinforcement learning, which allows it to be used for any type of visual reinforcement learning task (e.g., robot selecting actions to achieve a particular goal) instead of image recognition or classification. Nevertheless, supervised learning is a simple special case of this approach.

The actions a_t correspond to choosing the choosing the class label with the use of a softmax prediction. The reward r_t in the tth time-stamp might be 1 if the classification is correct after t time-stamps of that roll out, and 0, otherwise. The overall reward R_t at the tth time-stamp is given by the sum of all discounted rewards over future time stamps. However, this action can vary with the application at hand. For example, in an image captioning application, the action might correspond to choosing the next word of the caption.

The training of this setting proceeds in a similar manner to the approach discussed in section 11.6.2 of Chapter 11. The gradient of the expected reward at time-stamp t is given by the following:

$$\nabla E[R_t] = R_t \nabla \log(\pi(a_t)) \tag{12.1}$$

Backpropagation is performed in the neural network using this gradient and policy rollouts. In practice, one will have multiple rollouts, each of which contains multiple actions. Therefore, one will have to add the gradients with respect to all these actions (or a mini-batch of these actions) in order to obtain the final direction of ascent. As is common in policy gradient methods, a baseline is subtracted from the rewards to reduce variance. Since a class label is output at each time-stamp, the accuracy will improve as more glimpses are used. The approach performs quite well using between six and eight glimpses on various types of data.

12.2.2 Attention Mechanisms for Image Captioning

In this section, we will discuss the application of the visual attention approach (discussed in the previous section) to the problem of image captioning. The problem of image captioning is discussed in section 8.7.2 of Chapter 8. In this approach, a single feature representation $\overline{v}$ of the entire image is input to the *first time-stamp* of a recurrent neural network. When a feature representation of the entire image is input, it is only provided as input at the first time-stamp when the caption begins to be generated. However, when attention is used, we want to focus on the portion of image that corresponds to the word being generated. Therefore, it makes sense to provide different attention-centric inputs at different time-stamps. For example, consider an image with the following caption:

"*Bird flying during sunset.*"

The attention should be on the location in the image corresponding to the wings of the bird while generating the word "*flying*," and the attention should be on the setting sun,

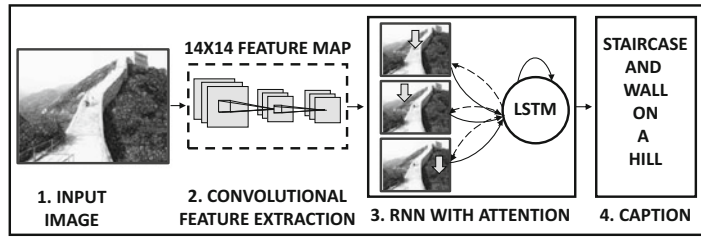


Figure 12.3: Visual attention in image captioning

while generating the word "*sunset*." In such a case, each time-stamp of the recurrent neural network should receive a representation of the image in which the attention is on a specific location. Furthermore, as discussed in the previous section, the values of these locations are also generated by the recurrent network in the previous time-stamp.

Note that this approach can already be implemented with the architecture shown in Figure 12.2 by predicting one word of the caption in each time-stamp (as the action) together with a location l_t in the image, which will be the focus of attention in the next time-stamp. The work in [562] is an adaptation of this idea, but it uses several modifications to handle the higher complexity of the problem. First, the glimpse network does use a more sophisticated convolutional architecture to create a 14×14 feature map. This architecture is illustrated in Figure 12.3. Instead of using a glimpse sensor to produce the modified version of the image in each time-stamp, the work in [562] starts with L different preprocessed variants on the image. These preprocessed variants are centered at different locations in the image, and therefore the attention mechanism is restricted to selecting from one of these locations. Then, instead of producing a location l_t in the $(t-1)$th time-stamp, it produces a probability vector $\overline{\alpha}_t$ of length L indicating the relevance of each of the L locations for which representations were preprocessed in the convolutional neural network. In hard attention models, one of the L locations is sampled by using the probability vector $\overline{\alpha}_t$, and the preprocessed representation of that location is provided as input into the hidden state h_t of the recurrent network at the next time-stamp. In other words, the glimpse network in the classification application is replaced with this sampling mechanism. In soft attention models, the representation models of all L locations are averaged by using the probability vector $\overline{\alpha}_t$ as weighting. This averaged representation is provided as input to the hidden state at time-stamp t. For soft attention models, straightforward backpropagation is used for training, whereas for hard attention models, the REINFORCE algorithm (cf. section 11.6.2 of Chapter 11) is used. The reader is referred to [562] for details, where both these types of methods are discussed.

12.2.3 Soft Image Attention with Spatial Transformer

We have already discussed the notion of *channel re-weighting* for convolutional networks with SENets in section 9.4.6 of Chapter 9. In this section, we will discuss the notion of *spatial re-weighting* with the use of an attention mechanism, which is referred to as *spatial transformer* [229]. Beyond serving as an attention mechanism, this general approach allows various types of spatial transformations, such as rotation, scaling, and translation — it can also achieve spatial invariance in a selective and image-specific way (unlike max-pooling). The approach is modular, and it can inserted anywhere in the spatial layers of an existing convolution network (such as between each pair of convolution layers). It produces a con-

volution block of exactly the same size as its input, and can therefore be integrated into an existing convolutional neural network architecture with very little disruption to its overall architecture (just like SENet).

The approach relies on the fundamental *polar decomposition result* [6] in linear algebra. The basic principle is that *multiplying the 2-dimensional row vector of coordinates of the image pixels with a square* 2×2 *matrix performs some combination of rotation, reflection, and scaling on the coordinates.* Furthermore, translation can be simulated by adding a third dimension value of 1 to the row vector of pixel coordinates and multiplying with a 3×2 matrix instead (where the final row of the 3×2 matrix contains the translations along the two spatial directions). In other words, we need a total of $2 * 3 = 6$ values to learn how to transform the image. These six values are generated as outputs of a *localization network* in an image-specific manner so that any transformations or attention operations are image-specific. Consider a 28×28 matrix, in which the (i, j)th value is simply (i, j), corresponding to a grid of coordinates containing the 28×28 pixel values in one of 256 feature maps (cf. Figure 12.4). Even though we are using the word "pixels" for simplicity in this discussion, the transformations are often applied to values in hidden layers (where activations are not primitive pixels). Furthermore, the illustrated image transformation in the figure is only notional, as the approach may be applied to a feature map in the hidden layer without similar interpretability. Then, the row vector $[i, j, 1]$ is a transformed to real-valued 2-dimensional coordinate mappings $[x_{ij}, y_{ij}]$ using multiplication with the 3×2 matrix A:

$$[x_{ij}, y_{ij}] = [i, j, 1]A \quad \forall i, j \in \{1, \ldots, 28\}$$

Stated simply, the pixel value in the integer coordinate $[i, j] \in 28 \times 28$ in the transformed image maps to the pixel value in the real-valued "coordinate" $[x_{ij}, y_{ij}]$ in the original image. In order to deal with the non-integer values of $[x_{ij}, y_{ij}]$, one can interpolate from the four nearest pixel coordinates surrounding $[x_{ij}, y_{ij}]$ in the original image. The new pixel value at $[i, j]$ is computed as a weighted average of the (typically) four pixel values surrounding "coordinate" $[x_{ij}, y_{ij}]$ by using the following (sub)-differentiable interpolation function:

$$h'_{ijq} = \sum_{m=1}^{28} \sum_{n=1}^{28} \max\{0, 1 - |x_{ij} - m|\} * \max\{0, 1 - |y_{ij} - n|\} h_{mnq} \quad \forall \text{ channels}$$
$$q, i, j \in \{1, \ldots, 28\}$$

Here, h_{mnq} is the value of the (m, n)th pixel in the original image, and h'_{ijq} provides the value of the $[i, j]$th pixel in the transformed image (for a particular channel q). If the set of coordinate pairs $[i, j] \in 28 \times 28$ map to a set of coordinate pairs $[x_{ij}, y_{ij}]$ that do not fully cover the original 28×28 grid, a part of the image will get cropped out as irrelevant.

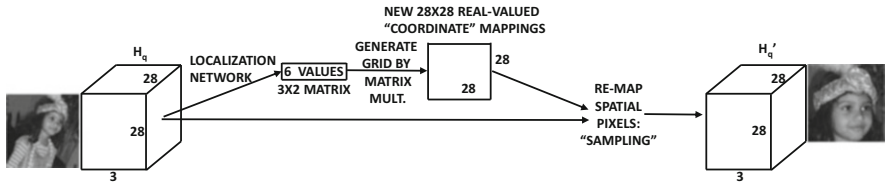


Figure 12.4: The spatial transformer module [Compare with channel-wise attention module of Figure 9.12]

Therefore, one way of zeroing on into a specific portion of an image is by using a diagonal matrix in the upper 2×2 submatrix of A with diagonal value of $s \in (0, 1)$ in both entries. The value of s provides an inverse magnification factor for the image. The lowest row $[a, b]$ of A translates the center of *attention* of this magnification so that the relevant portion of the image is retained. If rotations and disproportional distortions are not desired, the localization network can be made to output three values $[s, a, b]$ instead of 6, where the output s is created by a sigmoid activation function. One can understand the geometric nature of the transformation performed by A by considering the *polar decomposition* of its upper 2×2 submatrix A' into orthogonal matrix U and symmetric positive semidefinite matrix S as follows:

$$A' = US$$

Multiplying $[i, j]$ with A' can be considered a sequence of multiplications with U and S respectively. Multiplication with U rotates and/or reflects the image and multiplication with symmetric positive semi-definite matrix S magnifies the image by different scale factors along two mutually perpendicular directions (defined by its eigenvectors). The scale factors along the two eigenvector directions are the reciprocals of the corresponding eigenvalues (since we are inverse-mapping the transformed coordinates to the original ones). The lowest row of the original matrix A provides the translations along the two directions.

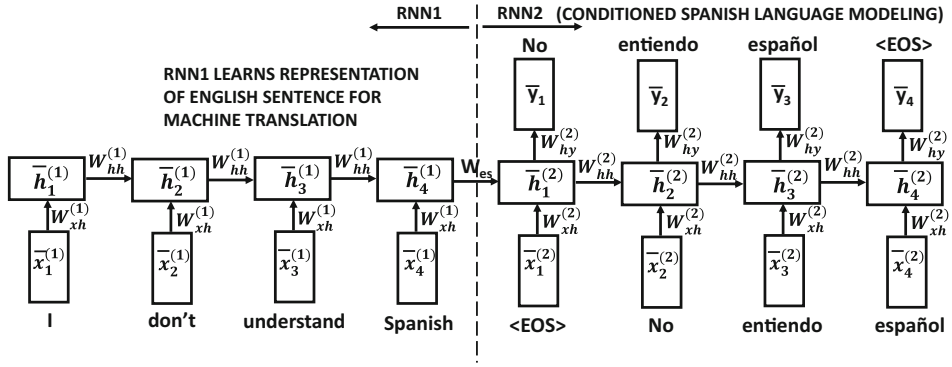
The 6 entries of the 3×2 matrix A are generated as a 6-dimensional outputs of a localization (neural) network performing regression. This network can be convolutional network or even a fully connected network. Note that the matrix A is therefore generated differently for each image, as each image has a different relevant transformation or "important" portion to be attended to. The overall architecture of the attention mechanism is shown in Figure 12.4.

12.2.4 Attention Mechanisms for Machine Translation

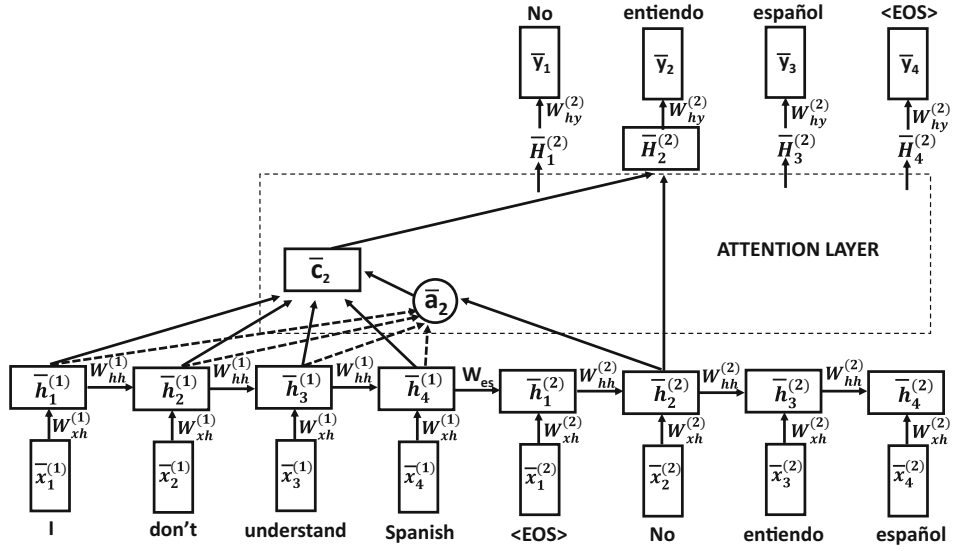
As discussed in section 8.7.3 of Chapter 8, recurrent neural networks (and especially their variants like LSTMs and GRUs) are used frequently for machine translation. In the following, we use a vanilla recurrent neural network with a single layer for simplicity in exposition (and ease in illustration of neural architectures), although more sophisticated architectures like LSTMs and GRUUs are used in practice. When multiple hidden layers are used, the attention mechanism is almost always applied to the top hidden layer. The two most common forms of RNN-based attention in neural machine translation are *Luong attention* [311] and *Bahdanau attention* [19]. Here, we present Luong attention.

Luong Attention

We start with the architecture discussed in section 8.7.3 of Chapter 8. For ease in discussion, we replicate the neural architecture of that section in Figure 12.5(a). Note that there are two recurrent neural networks, of which one is tasked with the encoding of the source sentence into a fixed-length representation, and the other is tasked with decoding this representation into a target sentence. The hidden states of the source language (encoder) and target language (decoder) networks are denoted by $h_t^{(1)}$ and $h_t^{(2)}$, respectively, where $h_t^{(1)}$ corresponds to the hidden state of the tth word in the source sentence, and $h_t^{(2)}$ corresponds to the hidden state of the tth word in the target sentence. These notations are borrowed from section 8.7.3 of Chapter 8.



(a) Machine translation without attention



(b) Machine translation with attention

Figure 12.5: The neural architecture in (a) is the same as the one illustrated in Figure 8.10 of Chapter 8. An extra attention layer has been added to (b).

In attention-based methods, the hidden states $h_t^{(2)}$ are transformed to enhanced states $H_t^{(2)}$ with some additional processing from an *attention layer*. The goal of the attention layer is to incorporate context from the source hidden states into the target hidden states to create a new and enhanced set of target hidden states.

In order to perform attention-based processing, the goal is to find a source representation that is close to the current target hidden state $h_t^{(2)}$ being processed. This is achieved by using the similarity-weighted average of the source vectors to create a context vector $\overline{c}_t$:

$$\overline{c}_t = \frac{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)}) \overline{h}_j^{(1)}}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} \tag{12.2}$$

Here, T_s is the length of the source sentence. Attention mechanisms couch this type of similarity weighting in terms of the language of database query processing. At any given

time stamp t (i.e., for the tth word), the state $\overline{h}_t^{(2)}$ being processed in the target language is treated as a query over the "database" of all states (words) in the source language to identify the best equivalent word (context). Like any database, the source language "database" has a set of *values-key pairs*, where the jth value and key in the source language "database" are both equal to $\overline{h}_j^{(1)}$. A *query* representation $\overline{h}_t^{(2)}$ in the target sentence creates a context vector, $\overline{c}_t$, by "soft-searching" for the best matching *key* $\overline{h}_j^{(1)}$ in the source sentence and presenting its *value*. By *soft-searching* we refer to the fact that rather than presenting a single best matching source-language state, the approach weights the corresponding values $\overline{h}_j^{(1)}$ of these keys using the query-key similarities (which can be interpreted as probabilities of best matching). This type of approach is a *cross-attention mechanism* as the similarity values are computed *across* two different sets of objects (hidden representations at different positions of source and target sentences). The similarity value used for weighting is the *attention weight* $a(t, s)$, which indicates the importance of (key) source vector at position s to (query) target vector at position t:

$$a(t,s) = \frac{\exp(\overline{h}_s^{(1)} \cdot \overline{h}_t^{(2)})}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} \tag{12.3}$$

These similarity weights sum to 1 like probabilities, and the entire vector of weights $[a(t,1), a(t,2), \ldots a(t,T_s)]$ for target $\overline{h}_t^{(2)}$ is denoted by the *attention vector* $\overline{a}_t$. The attended representation of the target position t is the expectation over *all* source hidden vectors $\overline{h}_j^{(1)}$ with probabilistic attention weights $a(t,j)$ for fixed t and varying j:

$$\overline{c}_t = \sum_{j=1}^{T_s} a(t,j)\overline{h}_j^{(1)} \tag{12.4}$$

Here, $\overline{c}_t$ provides the matching source-centric context for $\overline{h}_t^{(2)}$, which can be added to the information already contained in $\overline{h}_t^{(2)}$. Therefore, one can apply a single-layer feed-forward network to the concatenation of $\overline{c}_t$ and $\overline{h}_t^{(2)}$ to create a new target hidden state $H_t^{(2)}$:

$$\overline{H}_t^{(2)} = \tanh\left(W_c \left[\begin{array}{c} \overline{c}_t \\ \overline{h}_t^2 \end{array}\right]\right) \tag{12.5}$$

This new hidden representation $\overline{H}_t^{(2)}$ is used in lieu of the original hidden representation $\overline{h}_t^{(2)}$ for the final prediction. The overall architecture of the attention-sensitive system is given in Figure 12.5(b). Note the enhancements from Figure 12.5(a) with the addition of an attention layer. When translating "*I don't understand Spanish*," the attention mechanism is able to align relevant positions (e.g., *understand* and *entiendo*) in the English and Spanish versions of the sentence by returning high attention values for the corresponding state pairs.

This model is referred to as the *global attention model* in [311], because it uses all the words in the source sentence. The work in [311] also proposes a *local* attention model, which restricts the weighting to a subset of the most relevant words and can therefore be considered a combination of soft and hard attention. This approach focuses on a small window of context around the best aligned source state by using the importance weighting

generated by the attention mechanism. Such an approach is able to implement a limited level of hard attention without incurring the training challenges of reinforcement learning. The reader is referred to [311] for details.

Variations and Comparison with Bahdanau Attention

Several refinements can improve the basic attention model. First, the attention vector $\overline{a}_t$ is computed by exponentiating the raw dot products between $\overline{h}_t^{(2)}$ and $\overline{h}_s^{(1)}$, as shown in Equation 12.3. It is possible to increase the capacity of the model further by adding learnable parameters within the attention computation. The parameterized alternatives for computing similarity between source and target states were as follows [311]:

$$\text{Score}(t,s) = \begin{cases} \overline{h}_s^{(1)} \cdot \overline{h}_t^{(2)} & \text{[Dot product]} \\ (\overline{h}_t^{(2)})^T W_a \overline{h}_s^{(1)} & \text{[Parameterized dot product with } W_a\text{]} \\ \overline{v}_a^T \tanh\left(W_a \begin{bmatrix} \overline{h}_s^{(1)} \\ \overline{h}_t^2 \end{bmatrix}\right) & \text{[Parameterized FFN with } W_a \text{ and } \overline{v}_a\text{]} \end{cases} \tag{12.6}$$

The first of these options is identical to the dot product attention of Equation 12.3 and the second option (referred to as *general* in [311]) is its parameterized version. The third option applies a two-layer feed-forward network (FFN) to the concatenated source-target pair $[\overline{h}_s^{(1)}, \overline{h}_t^{(2)}]$ with the parameters of the two layers respectively set to matrix W_a and vector $\overline{v}_a$. This model was referred to as *concat* in [311] and as *additive attention* in the original Bahdanau paper where it was proposed [19] — it is, therefore, popularly referred to as *Bahdanau attention*. The parametrization of the last two models provides additional flexibility by enabling learning during training. The attention values can be computed from the vector of scores using softmax activation (as in the previous section):

$$a(t,s) = \frac{\exp(\text{Score}(t,s))}{\sum_{j=1}^{T_s} \exp(\text{Score}(t,j))} \tag{12.7}$$

These attention probabilities are used to compute the expected context vector $\overline{c}_t$ from the source sentence. Surprisingly, it was shown in [311] that the unparameterized dot-product was more accurate than parameterized alternatives in global attention models, although the parameterized alternatives did better in local attention models. It is possible that the good performance of the unparameterized dot product was a result of its regularizing effect on the model. The parameterized dot product model generally seemed to perform better than the *concat* model (i.e., additive or Bahadanau attention).

Finally, we briefly describe the original Bahdanau attention model by elucidating its differences from Luong attention. First, the encoder in Bahdanau attention is bidirectional and the decoder is uni-directional, whereas both are uni-directional in Luong attention. Therefore, each forward-backward state pair of the Bahdanau encoder was concatenated before additive attention computations. Second, for each decoder time-stamp t, the Bahdanau attention computation is applied to the target hidden state $\overline{h}_{t-1}^{(2)}$ from the *previous* time stamp to create context vector $\overline{c}_t$. This context vector is concatenated with $\overline{h}_{t-1}^{(2)}$ and *then* passed through one step of the recurrent neural network to create the current hidden state $\overline{h}_t^{(2)}$. The word prediction at t is made by passing this hidden state $\overline{h}_t^{(2)}$ through a deep output layer. Unlike Luong attention, Bahdanau attention does not have a separate attention layer to create a *post-attentive* hidden state such as $\overline{H}_t^{(2)}$, because its computation

flow is $\overline{h}_{t-1}^{(2)} \rightarrow (\overline{a}_t, \overline{c}_t) \rightarrow \overline{h}_t^{(2)}$, whereas that of Luong is $\overline{h}_t^{(2)} \rightarrow (\overline{a}_t, \overline{c}_t) \rightarrow \overline{H}_t^{(2)}$. Therefore, Luong attention is more modular and easily generalizable to an off-the-shelf architecture with the addition of an attention layer.

12.2.5 Transformer Networks

As evident from the machine translation example of the previous section, attention can be added to recurrent neural networks to improve accuracy. It turns out that can process sequences *without* using recurrent neural networks by combining attention mechanisms with *positional encodings*. This leads to a much simpler architecture with better performance. This approach drops the use of the recurrent neural network, and moves back to traditional multilayer neural network with important roles for attention and positional encodings. Such architectures are referred to as *transformer networks* [521]. The use of transformer networks has now become a worthy alternative to the recurrent neural network for natural language processing. One advantage of transformer networks over recurrent neural networks is that the underlying computations can be parallelized easily, which enables more efficient use of hardware advancements such as GPUs. The greatest weakness of the recurrent neural network architecture has been its resistance to parallelizeability as a result of its dependence on sequentially computed states; by doing away with this type of sequential computation, the transformer has a clear computational advantage. Gains in computational efficiency often translate to improved accuracy because of the ability to build larger models with more robust training.

12.2.5.1 How Self Attention Helps

The transformer learns embeddings of words (like *word2vec*) but it adjusts the embeddings for each specific usage of the word based both on its position and context within the sequence (or sentence). Consider a sequence of n words, each of which has codes $\overline{h}_1 \ldots \overline{h}_n$. These codes can either be derived from another learning mechanism like *word2vec* or learned/fine-tuned via backpropagation within the neural architecture of the transformer. While the representation $\overline{h}_j$ of the jth word in the sentence depends on its usage and context in the entire training data (from which it is derived), it is also desirable to adjust these representations based on the usage of the term in the specific sentence being processed by the transformer. This adjustment is achieved with the use of a *self-attention mechanism*, wherein the words in a sentence are used to provide additional context. In particular, the representations $\overline{h}_1 \ldots \overline{h}_n$ can be mapped to contextual representations $\overline{c}_1 \ldots \overline{c}_n$ by using dot-product self attention as follows:

$$\overline{c}_t = \frac{\sum_{j=1}^{n} \exp(\overline{h}_j \cdot \overline{h}_t)\overline{h}_j}{\sum_{j=1}^{n} \exp(\overline{h}_j \cdot \overline{h}_t)} \tag{12.8}$$

After this adjustment, the code for "*right*" in the sentence fragment "*turn right*" will be different from that in "*do right.*" While the recurrent neural network is also able to achieve contextual encoding for entire *sentences*, it seems to have greater problems with longer sentences such as the following: "*Selecting the right branch at the fork was a mistake, and I should have understood that it made better sense to choose the left one.*" This is because of the sequential updates of the states in the recurrent neural network in which each update scrambles some of the information in previous states. On the other hand, the transformer will be able to adjust the coding for "*right*" using attention from related words like "*left*" without regard to the distance between these two words.

One can also express this operation in terms of the relative similarity values a_{tj} between vectors $\overline{h}_j$ and $\overline{h}_t$:

$$a_{tj} = \frac{\exp(\overline{h}_j \cdot \overline{h}_t)}{\sum_{j=1}^{n} \exp(\overline{h}_j \cdot \overline{h}_t)} \tag{12.9}$$

The relative similarity values a_{tj} are positive and add to 1 over fixed t and varying j — these are used to create re-weighted codes as follows:

$$\overline{c}_t = \sum_{j=1}^{n} a_{tj} \overline{h}_j \tag{12.10}$$

The magnitude of the vanilla dot product tends to grow with the square-root of the dimensionality — therefore, it is common to use *scaled* dot product attention in which each $\overline{h}_j \cdot \overline{h}_t$ is divided by the square-root of the dimensionality of $\overline{h}_j$ before exponentiation.

12.2.5.2 The Self-Attention Module

The attention operation can be expressed using the notions of queries, keys, and values, which derives its motivation from the fact that attention adjusts word representations based on "matching" query words with related words in the sentence. For each *query* vector $\overline{h}_t$, it is converted into a corresponding attention-modified vector $\overline{c}_t$, which is a weighted average of *values* (value vectors) $\overline{h}_j$ contained in the sentence (see Equation 12.10). Each value vector is associated with a *key vector* with which queries are "matched" in a soft way with dot product attention. In this simplified case, the value and key vectors are the same. The query-specific "matching" weight of a value vector (for attention re-adjustment) is obtained by applying softmax to the dot product similarity between the query vector and the key vector of that value vector. Therefore, the notation $\overline{h}_t$ in Equation 12.9 is the query, and the notation $\overline{h}_j$ for $j \in \{1 \ldots n\}$ in the same equation refers to the key. By using this approach, the representation for a query word on a particular topic (e.g., sports) is more likely to be affected by the representations of other sports-related words in a sentence. Therefore, the new representation is more context-sensitive.

In this simplified exposition of attention, the queries, keys, and the values associated with word j have the same p-dimensional column vector representation $\overline{h}_j$, which begs the question as to why we need separate terminologies for different portions of the attention equation. First, it provides an analogy to query-matching weights for the attention mechanism. Second, in many applications of *cross-attention* like machine translation, the queries in a target language may be different from the key/value pairs in a source language. Finally, even in self-attention, it is common to transform queries, keys and values to *different* q-dimensional vectors $W^Q\overline{h}_j$, $W^K\overline{h}_j$, and $W^V\overline{h}_j$ using learned $q \times p$ matrices W^Q, W^K, and W^V, which results in greater model capacity. Consider the $p \times n$ matrix H obtained by stacking the column vectors $[\overline{h}_1 \ldots \overline{h}_n]$ adjacent to one another. The queries, keys, and values for a particular input sentence of length n can be respectively represented by $n \times q$ matrices $Q = (W^Q H)^T$, $K = (W^K H)^T$, and $V = (W^V H)^T$. It is also possible for the value matrix W^V to have a different size $q_1 \times p$, although the original transformer paper uses $q_1 = q$. The matrices W^Q, W^K, and W^V are considered a part of the attention subunit, and they are learned during backpropagation. This additional step is important to prevent the attention model from under-fitting and improve accuracy.

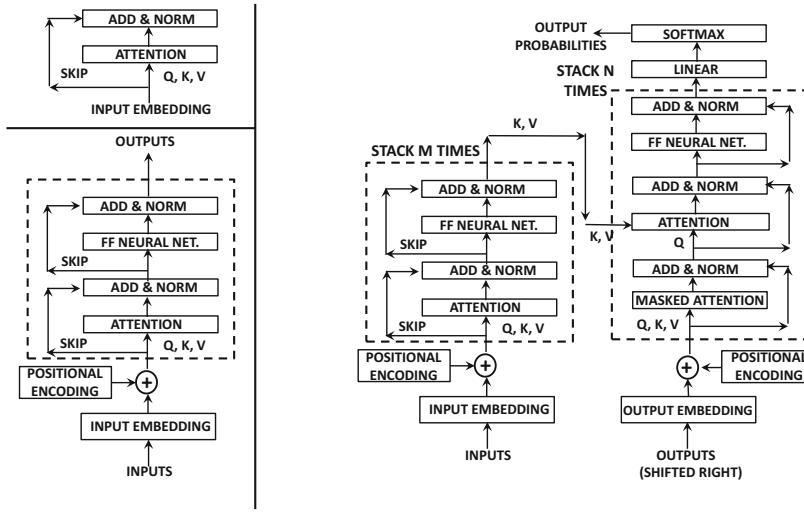
The attention operation (with scaling) can be implemented by applying the softmax operation to each row of $(QK^T)/\sqrt{q}$ to yield a matrix of the same size and then multiplying

with V to create the $n \times q$ re-weighted value matrix $\text{Softmax}(QK^T/\sqrt{q})V$. These operations can be implemented with efficient and parallelizeable matrix operations on GPUs. In order to ensure that the attention layer outputs a vector of the same p-dimensional size as the input $\overline{h}_j$ (and also to enable additional learning), a linear dimensionality-fixing transformation $W^{fix}\overline{x}$ using $p \times q$ matrix W^{fix} is applied to each transposed row $\overline{x}$ of the $n \times q$ matrix $\text{Softmax}(QK^T/\sqrt{q})V$. This entire sequence of attention-centric transformations from p-dimensional word vector embeddings $\overline{h}_1 \ldots \overline{h}_n$ to another set of p-dimensional embeddings is referred to as the Att-Sublayer.

In order to avoid losing any important information in the original representation $\overline{x}$ of a particular word when it is converted to Att-Sublayer($\overline{x}$) with the use of the attention sublayer, a skip-level connection is used in which the attention-modified vector is added to the original vector and then layer-normalized (cf. section 8.3.1) to create the normalized output $\overline{x}' = \text{LayerNorm}(\overline{x} + \text{Att-Sublayer}(\overline{x}))$. This principle is similar to that of residual learning popularly used in the gating mechanisms of recurrent neural networks (via LSTMs) and convolutional neural networks (via ResNets) to avoid losing features that have already been learned. The architecture for this basic attention-based mechanism is shown in the upper portion of Figure 12.6(a). Note that the queries, keys, and values (denoted by Q, K, and V) are shown as inputs to this attention block for simplification, although they are technically created by this block using the learned matrices W^Q, W^K, and W^V on the input matrix H into the block, and therefore these matrices belong to the attention block as well. We refer to the overall function computed by this block as $[\overline{h}'_1 \ldots \overline{h}'_n] = \text{SelfAtt}(\overline{h}_1, \ldots, \overline{h}_n)$.

Finally, the p-dimensional attended vectors are subjected to a two-layer feed-forward network, with hidden layer dimensionality of $4p$ and output dimensionality of p (i.e., same as that of input). Therefore, this feed-forward network $\text{FFN}(\cdot)$ will contain $p \times 4p$ and $4p \times p$ matrices W_1 and W_2, which are also learned during backpropagation:

$$\text{FFN}(\overline{x}) = W_1(\text{ReLU}(W_2\overline{x}))$$



(a) Basic self-attention function $\text{SelfAtt}(\overline{h}_1, \ldots, \overline{h}_n)$ (top) and encoder (bottom)

(b) Full sequence-to-sequence transformer with decoder hooked to encoder

Figure 12.6: The Transformer Architecture

As in the case of the attention sublayer, the output $\text{FFN}(\overline{x})$ of the feed-forward sublayer is subjected to skip-connections and normalization to create the output $\text{LayerNorm}(\overline{x} + \text{FFN}(\overline{x}))$. The combination of a self-attention sublayer and the feed-forward s sublayer creates a single "layer" of the transformer (demarcated by a dotted line in Figure 12.6(a)), and its output dimensionality is unchanged from its input. A total of M such layers are stacked in order to create a *deep encoder* with more expressive learning abilities. The p-dimensional output of a layer (denoted above by $\text{LayerNorm}(\overline{x} + \text{FFN}(\overline{x}))$) for each of the n words in the sequence can be used to create another $p \times n$ matrix H', which can be input to the next layer and transformed with the matrices W^Q, W^K, W^V, to create new query, key, and value matrices of the next layer. The parameters of the different layers are not shared. Therefore, while we use shared notations such as W^Q, W^K, W^V, W^{fix}, W_1, and W_2 repeatedly across layers to reduce notational complexity, these matrices are different in each layer. The original transformer [521] used $M = 6$, although recent language models use much deeper architectures. The encoder processes each word in the sentence independently and will therefore create an output code for each word in the input sequence. This is different from a traditional recurrent neural network that creates fixed-length representations of full sentences (which can be suboptimal). This entire process represents the *encoding* portion of a sequence-to-sequence learner (see bottom portion of Figure 12.6(a)), although we have yet to discuss the creation of the positional encodings depicted in this diagram.

12.2.5.3 Incorporating Positional Information

The attention mechanism does not use the relative positions of the words while computing contextual representations. Note that the vectors $\overline{h}_1 \ldots \overline{h}_n$ are processed independently, and changing the order of words in a sentence does not change the learning in any way. Therefore, unlike the encoding of a recurrent neural network, the encodings of the transformer will be the same even after sequence permutation. This can be a problem because the ordering of the words in phrases like "*the hot dog fell on the pavement*" and "*the dog fell on the hot pavement*" can be very significant. Therefore, the transformer adds *positional encodings* to the word codes *before* performing any other operation. Like the word embedding of dimensionality p, the positional encoding is a p-dimensional vector $\overline{z}_t$ for the word in the tth position. Then, the jth component z_t^j of vector $\overline{z}_t$ is defined as follows:

$$z_t^j = \begin{cases} \sin(t \cdot \omega_j) & \text{if } j \text{ is even} \\ \cos(t \cdot \omega_{j-1}) & \text{if } j \text{ is odd} \end{cases}$$

The value of the frequency ω_j is defined as $\omega_j = 10000^{-(j/p)}$. The basic idea here is to capture the positional information in sinusoidal waves with varying periods (corresponding to varying levels of granularity). Positional vector components that are close to one another at any particular level of granularity will have similar values for the positional encoding. The positional encodings are then added to the word encodings in order to create encodings that incorporate positional information:

$$\overline{h}_t \Leftarrow \overline{h}_t + \overline{z}_t$$

The initial word embeddings $\overline{h}_t$ (to which positional encodings are added) are obtained by multiplying a parameter matrix containing encodings for each word with one-hot encoded input vectors for each word in the sequence. This weight matrix (at the bottom of Figure 12.6(a)) can be initialized to *word2vec* encodings and fine-tuned during backpropagation.

12.2.5.4 The Sequence-to-Sequence Transformer

As in any sequence-to-sequence learning method, one needs a decoder along with the encoder. Consider the case of the machine-translation application, in which we are trying to translate the sentence "*I don't understand Spanish*" into "*No entiendo espanol*". In Figure 12.6(b), we have shown the decoder on the right. It is immediately evident that the structure of the decoder is very similar to that of the encoder. The decoder layers stacked N times, just as the encoder layers are stacked M times. However, both self-attention and cross-attention are used in different parts of the decoder in order to enable translation.

The *entire source sentence* is input to the encoder, whereas only the leftmost portion of the target sentence is input into the decoder in order to output the probability of the word occurring after this segment. This process is repeated for each word, and therefore the decoder requires a training step for each word in the sentence (which makes it slower). As an example, the encoder is always taking as input the full English sentence "*I don't understand Spanish*," but it takes as input only the leftmost portion such as' "*<START> No*" or '*<START> No entiendo*" When "*<START> No*" is input, the goal is to predict the probability of the next word "*entiendo*" without any awareness of the occurrence of this word and "*espanol*" in the sentence. This is achieved with a masking mechanism, so that words occurring after the current prediction word are forced to have an attention weight a_{tj} of 0 during the first self-attention round. A "*<START>*" tag is added to the beginning of each target sentence to shift it right by one unit. Therefore, when "*<START>*" is input, the goal is to predict "*No.*"

After passing the target sentence (fragment) through the self-attention layer, one applies a second cross-attention mechanism. In this case, the queries are defined by the words of the input target sentence (segment), and the keys/values are defined from the representation of the source sentence produced by the encoder. This point is highlighted in Figure 12.6(b) by the fact that the pair (K, V) is created by multiplying the transformations W^K and W^V with the encoder input matrices, whereas the query matrix Q (representation matrix of words in incompletely translated sentence) is obtained by multiplying W^Q to the decoder input matrices. This type of cross-attention mechanism gives greater weights to the most relevant words in the source sentence while translating the next word, which is similar in principle to the RNN-based cross-attention computation in section 12.2.4. The difference is that the translation model no longer uses a recurrent network — attention is all that is needed along with positional encodings. The decoder blocks are similar to encoder blocks in that they contains feedforward networks, skip connections, and normalizations. The decoder also contains multiple layers like the encoder. After passing each word in the partially translated sentence through the decoder, their vectors are passed through a linear layer (or averaging layer) to create a single code, and the softmax function is applied on the similarity between this code and the language dictionary codes in order to predict the probability of the next target word in the translated language. The predicted word is compared to the ground-truth word with cross-entropy loss for training.

12.2.5.5 Multihead Attention

As discussed earlier, the attention mechanisms need not be applied to the original codes corresponding to queries keys and values, but to transformed vectors $W^Q\overline{h}_j$, $W^K\overline{h}_j$, and $W^V\overline{h}_j$, where W^Q, W^K, and W^V are $q \times p$ matrices. It is typical for q to be less than p, which results in *projections* to smaller vectors. These *projection* matrices are learned parameters within the neural architecture. The projection matrices focus on specific dimensions (meta-concepts) of the words, and the effect is to focus attention on these meta-concepts.

A single word may contain multiple meta-concepts of interest. The point of the multi-head mechanism is that we can use k different sets of such projection matrices (denoted by W_i^Q, W_i^K and W_i^V) in parallel in order to learn different output vectors for the words in the same input sequence. Therefore, the attention mechanism can focus on different sets of meta-concepts. Each of these parallel pipelines is an *attention head*. The q-dimensional output vectors of these different attention heads are then concatenated into one long vector of length $k \cdot q$ that contains the information about all the different meta-concepts. However, since $k \cdot q$ may be different from p, one needs to multiply this vector with the $p \times (k \cdot q)$ matrix W^{fix}, which is also a part of the attention block as a linear feed-forward unit. This modification with multiple attention heads is performed to the basic attention block at the top of Figure 12.6(a), and so all matrices such as W_i^Q, W_i^K, W_i^V, and W^{fix} belong to that attention block (without sharing across layers). The matrices W_i^Q, W_i^K, W_i^V, and W^{fix} can be learned during the backpropagation of the end-to-end model of sequence-to-sequence learning. Multihead attention is more robust and represents the practical implementation of the transformer network. It is also highly parallelizeable, since different heads can be implemented on different GPUs or machines. Although multiple attention heads do improve performance, an interesting observation [335] is that most layers require only a single head for good performance, and the redundant heads can be pruned either in a validation phase or (more efficiently) by estimating the sensitivity of the loss function to each head via the backpropagation algorithm.

12.2.6 Transformer-Based Pre-trained Language Models

Transformers have fundamentally changed the landscape of natural language processing by gradually replacing recurrent neural networks across the board. One mechanism to achieve this goal is to create a large-scale, pre-trained language model (that works off an internet-scale corpus); two examples include OpenAI's *Generative Pretrained Transformer (GPT-n)* [50], and Google's *Bidirectional Encoder with Transformer (BERT)* [101]. However, an important difference of these models from the transformer model of the previous section is that GPT-n and BERT are more generic models for unsupervised learning and language modeling rather than machine translation — such unsupervised models can be fine-tuned for a large number of natural language processing tasks and they use *either* the encoder or the decoder of the transformer. Therefore, the basic architecture of these models is somewhat different from that of the vanilla transformer architecture, which has a connected encoder-decoder pair. Aside from the fact that the encoder feeds into the decoder in the vanilla transformer, the main difference between the encoder and decoder is that the former is bidirectional, whereas the latter is unidirectional (with the use of masked inputs only in one direction). GPT-n uses only the decoder portion of the architecture (thereby inheriting its unidirectional nature), whereas BERT uses the encoder (with its bidirectional characteristics). On the other hand, T5 [400] uses the encoder-decoder structure of the original transformer, and is therefore a text-to-text model. In the case of decoder use (GPT-n), cross-attention units are replaced with self-attention units, whereas in the case of encoder use (BERT), the outputs to the decoder now feed into an additional output layer predicting words at some positions. All these models are massive and use large training data sets with hundreds of billions of parameters. The effect of training data and model size has been truly remarkable in terms of learning the ability to perform a wide variety of language processing tasks.

12.2.6.1 GPT-n

The GPT-n series of architectures has had several editions, which are GPT, GPT-2, GPT-3, GPT-3.5, GPT-3.5 Turbo, and GPT-4, each of which has been an improvement over the previous one (typically by increasing the scale of the training data and by making small architectural modifications). Among the various models, technical details are available only about GPT-3 and earlier models. Even the number of parameters of the latest models are not known beyond the fact that they are significantly larger than the 175 billion number for GPT-3. Aside from the fact that GPT-3.5, GPT-3.5 Turbo, and GPT-4 use a larger number of parameters in the context of similar but larger transformer architectures (as compared to GPT-3), the versions of GPT starting with GPT-3.5 Turbo also use reinforcement learning with human feedback. It is noteworthy that reinforcement learning has become a training mechanism in later versions of GPT not only for pre-training but also for fine-tuning of specific applications of GPT like ChatGPT. Although a technical report[1] of GPT-4 is available, it focuses on only the capabilities of GPT-4, and it does not provide specific technical details of the reinforcement learning algorithm or the specific way in which the transformer architecture is expanded to use a larger number of parameters. According to the aforementioned technical report on GPT-4, this restriction on dissemination of technical details is in view of the "competitive landscape" of such AI models. What is known is that GPT-4 has a transformer portion of the architecture, which is similar to (but larger than) GPT-3, but it has additional training mechanisms based on reinforcement learning with human feedback. Beyond the fact that the proximal policy optimization (PPO) [453] was used for reinforcement learning, no further details were provided on the algorithmic specifics. Nevertheless, the transformer continues to be the primary portion of the architecture of GPT-4, which is also the approach used in GPT-3. The main difference between GPT-3 and GPT-4 from the neural architecture point of view seems to be the scale of the model. Therefore, this section will focus on the technical description of the transformer architecture, as it appears in GPT-3 (since this is the last version of the model for which precise technical details are available). GPT-n uses the decoder portion of the sequence-to-sequence architecture of Figure 12.6(b) both for training and for prediction. Since there is no cross-attention module, it is replaced with self-attention inputs of key-value pairs. Like all practical transformer architectures, a multihead version is used.

Unlike the original transformer architecture for machine translation, GPT-n learns a language model from the data by predicting the next word, given all the words to the left of the current word. Therefore, the input to GPT-n is a sequence in which all words to the right of the target word are masked. The output is an array of softmax probabilities of next-word prediction. GPT-3 uses a maximum sequence input length of 2048 and pads shorter sequences. Using all distinct words in the vocabulary as tokens can blow up vocabulary size. Therefore, GPT-3 uses *byte-pair encodings* [459] where subwords of two characters are used as tokens instead of entire words. For example, instead of "love," the tokens "lo" and "ve" are used. Since a byte reflects $2^8 = 256$ possible characters, a byte-pair encoding has a maximum vocabulary size of $256^2 = 65536$. However, *merge rules* are used to select about $50,000$ byte-pairs by merging the most frequently co-occurring character pairs. The original 256 character tokens and special end-of-text tokens are also included as GPT-3 tokens. These tokens are then converted into real-valued embedding vectors of dimensionality 12288 in GPT-3, much as *word2vec* converts words to vectors. GPT-n creates these vectors by learning the one-hot encoded token to embedding transformation during backpropagation. Therefore, the overall dimensionality of the embedded input sequence is 2048×12288. Throughout this section,

[1] https://arxiv.org/abs/2303.08774.

we will use "word" and "token" interchangeably, even though a GPT-3 (or BERT) token is not really a word from the semantic perspective.

As in the decoder of Figure 12.6(b), positional encodings are added to the sequence and then self-attention modules are used along with feed-forward layers in order to create context-sensitive word embeddings. Unlike the decoder in the original transformer architecture, cross-attention is not used (since there is no encoder input to decode). The latest (GPT-3) model adopts the *sparse transformers* [71] variant among its layers. The feed-forward network of GPT-3 contains a single hidden layer with $4 * 12888$ units. The largest GPT-3 model contained 96 self-attention layers with 96 attention heads of dimensionality 128. The softmax output is trained with cross-entropy loss in order to learn the weights of the model. Aside from Wikipedia, Books1, Books2, and WebText2 datasets, GPT-3 uses a very large common crawl of the Web (via the Common Crawl dataset), which is filtered for quality to reduce the effect of contamination. The Common Crawl data set contains 410 billion tokens out of the total of 499 billion tokens in the full training data used by GPT-3. The details of these datasets are discussed in [50].

A notable characteristic of GPT-n is that it uses the same pre-trained model for all downstream tasks. This is possible because next-word prediction is a fundamental language modeling task with high reusability. The model can be optionally fine tuned further with some application-specific examples. Note that one can easily use a language model for the original transformer application of machine translation as long as the concatenated pair of sentences in the two languages is treated as a single sequence. However, a few examples of sentences in the two languages help GPT-3 significantly to the point of reaching state-of-the-art performance.

An interesting application of GPT-3 is a conditional generative model that creates near-human-quality text by providing it contextual input such as the prompt "Article:", a title, and a subtitle. GPT-3 is able to write short articles of about 200 words that cannot be easily distinguished from human-written articles. In addition GPT-3 was able to achieve state-of-the-art performance on several closed-book question-answering tasks. GPT-3 performed well on some common-sense reasoning on tasks and poorly on others. Overall, GPT-3 was able to able to match or outperform a majority out of twelve natural language processing tasks. This is particularly impressive, considering the generic nature of the model. The performance on arithmetic tasks was particularly poor, as it achieved only 9%-accuracy on five-digit operations. This is not particularly surprising, as it learns a language model from text (rather than arithmetic computation), and rarely encounters all pairs of five digit numbers in sequences. Yet, the 9% accuracy is much higher than the type of learning in a language model would indicate. This suggests that GPT-3 is learning at least some common patterns in arithmetic operations. However, the poor performance in reasoning and arithmetic strongly suggests that the level of "understanding" in the model is limited to semantic patterns inherent in languages. The performance of GPT-4 is much more impressive on a wide variety of tasks. It scores well on most advanced placement tests, SAT, GRE, bar exams, and the US MLE (medical exam). It also does well at code completion. Aside from this, these models have been wrapped into applications like ChatGPT and InstructGPT, which have captured the imagination of the broader population. All these results portend a promising future for large language models.

InstructGPT and ChatGPT

InstructGPT [375] and ChatGPT [682] are two applications that combine the fine-tuning of GPT-n with reinforcement learning to create agents that have the ability to instruct

and chat with users. InstructGPT was constructed first, and ChatGPT was built as an enhancement. At this time, the specific details of only InstructGPT are available as a white paper [375]. Even in the case of InstructGPT, precise technical details have not been provided. Therefore, the following discussion is presented at a high level.

As the name suggests, InstructGPT is designed to follow natural language instructions and provide the steps for a task such as the following: "*Write a guide on how I can cook a dish of chicken lasagna.*" The main innovation of InstructGPT is to align language models with user intent by using human feedback. The InstructGPT model uses a combination of supervised learning and reinforcement learning. A set of user prompts are used to collect a dataset of labeler demonstrations for fine-tuning GPT-3 using supervised learning. The user-defined rankings of model outputs are then used to fine-tune this supervised model using reinforcement learning from human feedback. A reward model is trained to predict which model output would be preferred by labelers. In particular, the reward is maximized using the proximal policy optimization algorithm [453]. By using this approach, toxic outputs are avoided by the guardrails of using carefully screened labeling and feedback.

ChatGPT is a similar application of InstructGPT, which is able to perform chats with humans using a Web interface [682]. ChatGPT uses GPT-3.5 Turbo and beyond as its pretrained model. To date, it is one of the most realistic chat applications ever developed using artificial intelligence. The broad approach to training ChatGPT is similar to that of InstructGPT in the sense that it uses a combination of supervised and reinforcement learning. The proximal policy optimization algorithm is used for reinforcement learning, as in the case of InstructGPT. In this case, human AI trainers played both sides in conversations between the user and an AI assistant for generating the training data for reinforcement learning. The trainers were provided access to written suggestions to model their responses. This new dialogue dataset was used after transforming it into a dialogue format.

In order to create the reward model, the comparison data contained two or more model responses ranked by quality. The responses were collected using conversations between AI trainers and the chat application. Several alternative completions of a sampled message were ranked by AI trainers in order to generate training feedback. The reward models were leveraged to fine-tune it using proximal policy optimization [453]. After several iterations of this process, a surprisingly powerful model was obtained that could create realistic chats and also continue to provide instructions like InstructGPT.

12.2.6.2 BERT

GPT-n uses a uni-directional language model, which can pose challenges in some tasks like question-answering, where context from both directions is useful. For example, the embedding of the word "*right*" is different in "*right turn*" and "*right choice*," and disambiguation of "*right*" can only be performed by examining the portion occurring after it. Unlike GPT-n, BERT adapts the encoder of the transformer in Figure 12.6(b) rather than the decoder used by GPT-3; this causes the training process to be bidirectional. BERT uses a stack of either 12 or 24 layers (instead of the 6 layers used in the original transformer). The tokenization approach in BERT uses *WordPiece* [556], which is similar to byte-pair encodings, except that it maximizes the likelihood of the training data in order to perform merges (rather than using frequent co-occurrence). The positional embeddings are learned in BERT [532] rather than fixed sinusoidal functions.

BERT uses a loss function that is a combination of missing word prediction and sentence contiguity prediction. Therefore, each training example is a pair of sentences in order to aid with both types of prediction. For the missing word component, about 15% of the words in

the input are randomly chosen for prediction, of which 80% are replaced with [MASK] in the input, and the remaining are equally likely to be replaced by the true word or a randomly chosen word (like a de-noising autoencoder). BERT places a word-prediction layer at the end of the encoder to associate a loss with the aforementioned predictive positions. When the embedding of each word has been produced by the final layer of the deep transformer (encoder), the dot product of each of these representations with the input representation of each lexicon word is computed and converted to a probability value with the softmax function. This probability is used to compute the cross-entropy loss with respect to the true word.

The second component of the loss is based on *next-sentence prediction*. The model is fed pairs of sentences, and the task is to predict whether these pairs occur consecutively in a document. Therefore, 50% of the training pairs occur consecutively in the document, and the other 50% are randomly selected. A [SEP] token is added at the end of each sentence, and a special [CLS] token is added at the beginning of the first sentence. An embedding for "Sentence A" is added to first sentence and that for "Sentence B" is added to the second sentence of the pair, because it helps the model distinguish between the two sentences beyond just the use of the [SEP] token. The encoder output of the [CLS] token is passed through the classification layer to output a probability of whether or not the two sentences are consecutive. Logarithmic loss is applied to this probability value. The two loss components of missing word and sentence contiguity prediction are then aggregated into a single value. The first loss component learns semantic relationships across words, whereas the second loss component learns semantic relationships across sentences. BERT uses all of the text portion of English Wikipedia and BooksCorpus [593].

Another difference between BERT and GPT is that BERT needs an additional task-specific neural layer or changed loss function and corresponding fine tuning to use it for specific applications. This difference is necessitated by the fact that BERT does not use a wide-spectrum language model for the training process (which can be easily adapted to tasks like machine translation without architectural changes). For token-wise tasks like tagging, labels are associated with output word embeddings, whereas for sentence-wise tasks, labels are associated with sentence-wise tokens like [CLS]. For sentence-level classification tasks, a single sentence can be input to BERT, with a [CLS] at the beginning and [SEP] at the end, which is followed by a *degenerate null sentence*. This is because BERT really expects to see sentence pairs rather than a single sentence, and the second sentence of the pair is simply a null sentence. The representation of [CLS] is then mapped to a class probability with an additional output layer. In general, the nature of the output layer varies with the task at hand, since token-level tasks will need to associated weights with the output representations of tokens. The output-layer weights need to be fine-tuned with the changed architecture and loss function (specific to the task at hand). The amount of fine tuning required is typically modest, and BERT seems to perform quite well on a variety of tasks such as sentiment analysis, question answering, and named entity recognition. Overall, BERT achieved state-of-the-art performance on eleven natural language processing tasks. Google search queries are now processed by BERT.

12.2.6.3 T5

The T5 model [400] uses a text-to-text model for pretraining, and it therefore follows the original transformer architecture more closely than the other two models. The basic idea behind using text-to-text models is that *sequence-to-sequence learners can be treated as multi-task learners by appending a task-specific prefix*. For example, if one wanted to translate from English to German, one can use the following input and output sequences:

Input Sequence: translate English to German: That is good.
Output Sequence: Das ist gut.

Note that it is possible to translate to a different language by using a different prefix. Similarly, for a *text summarization task*, the input and output sequences would be as follows:

Input Sequence: summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi. ⟨ Paragraph continues specifying number of hospitalized people and identity of county ⟩
Output Sequence: six people hospitalized after a storm in attala county.

The grammar checking task, which is often applied to the *CoLA* benchmark (i.e., *Corpus of Linguistic Acceptability*), is achieved by adding the prefix "cola sentence" to the input, and the output can be "acceptable" or "not acceptable."

Input Sequence: cola sentence: the course is jumping well.
Output Sequence: not acceptable

The aforementioned example provides an instance of sentence classification, which can be generalized to other natural language processing applications such as *sentiment analysis*. In order to support these types of diverse tasks, the T5 transformer uses *a single unsupervised pretraining* phase followed by task-specific fine-tuning. The pre-training approach is motivated by BERT and it uses masked language modeling on a large corpus (described at the end of this section). The basic idea is to drop out about 15% of the tokens and replace *consecutive* spans of dropped tokens with *sentinel tokens*. For example, consider the following sentence in which the crossed-out portions need to be masked:

Thank you ~~for inviting~~ me to your party ~~last~~ week.

Then, the input to the encoder of T5 removes the crossed-out segments, and replaces them with sentinel tokens that are unique to the specific sequence at hand. In this case, the two tokens $\langle X \rangle$ and $\langle Y \rangle$ are used for the two cross-out spans:

Thank you $\langle X \rangle$ me to your party $\langle Y \rangle$ week.

Note that since "for inviting" is a consecutive set of masked out tokens, it is replaced with a single sentinel token. The outputs then contain these sentinel tokens followed by the corresponding sequences that they represent in the original (unmasked) input. An additional sentinel token $\langle Z \rangle$ is reserved as the final token delimiting the end of the output. Note that all the sentinel tokens are different from one another in order to distinguish them. Therefore, the ground-truth sequence provided to the decoder of T5 during training is as follows:

$\langle X \rangle$ for inviting $\langle Y \rangle$ last $\langle Z \rangle$

The training of encoder-decoder architecture used 12 layers in both the encoder and the decoder. All sublayers and embeddings have a dimensionality of 768. The feed-forward networks in each block contained an intermediate layer with a dimensionality of 3072 followed by a ReLU nonlinearity. All attention mechanisms used 12 heads, each with an inner dimensionality of 64. For task-specific tuning, the (best) model uses a *gradual unfreezing approach*, wherein only the top block of the encoder and decoder were tuned first (with the task-specific data) while freezing the parameters in lower blocks. Then, the next lower blocks of the encoder-decoder stack were tuned together with the top blocks (while keeping

the lower 10 blocks frozen). This approach was continued all the way down the 12 blocks in the encoder-decoder stack.

Note that this approach uses pretraining followed by fine-tuning for a single task. It is also possible to perform true multi-task learning by mixing examples together from different tasks (during the fine-tuning phase). Since each task has its own prefix, the model learns to recognize the specific tasks based on the prefix in the input at prediction time. The main challenge with multi-task learning is that it is possible for the model to overfit to specific tasks, and the training data for some tasks may interfere with others. This problem is referred to as *negative transfer*. Therefore, the T5 transformer mixes examples roughly in proportional to their task-specific data set sizes (i.e., concatenating the data sets), while reducing the proportion of those data sets that are too large (to avoid crowding out the tasks with smaller data sets). This is achieved by placing a threshold on the maximum size of each data set. A variant of this approach uses *temperature-based mixing*, wherein the probability of sampling from a data set is proportional to $s^{1/t}$, where s is the data set size and t is the temperature parameter. At $t = 1$, this approach reduces to concatenating the data sets, and larger values of t resulted in a damping effect with respect to data set size (to avoid crowding out). The best results were obtained at $t = 2$, although the multi-task learning approach could not match the performance of the model that used single-task-specific fine tuning after pretraining. This difference is natural, because there is some level of cross-interference between the models of different tasks in multi-task learning.

The pre-training data that was used for building the masked model was *The Colossal Clean Crawled Corpus*, which leveraged the *Common Crawl* as its original source after cleaning it. The cleaning process removed text containing menus, error messages, gibberish, duplicate text, code, or offensive words. Lines that did not contain punctuation or contained fewer than three words were dropped. Any page with fewer than five sentences was removed. A number of other heuristics that were used to clean the data set are described in [400].

12.2.7 Vision Transformer (ViT)

Just as the text transformer processes sequences without using RNNs, the vision transformer [106] processes images without using CNNs. The vision transformer simply converts an image into a sequence of *patches*, and then uses the transformer model in a very similar way to BERT. Consider an $n \times n \times 3$ representation of an image with three channels corresponding to RGB colors. The vision transformer divides the spatial region into $n/p \times n/p$ patches of size $p \times p \times 3$. In other words, the rectangular image is divided into n^2/p^2 square tiles or patches. Each of these patches is treated in a similar way to a "word" in natural language processing, and a sequential ordering is imposed on these patches just as one orders English words written on a piece of rectangular paper. An example of show an image may be split into 3×3 tiles is shown in Figure 12.7. One can view the patch size as the degree of resolution at which the input is represented as a sequence. Reducing the patch size improves input resolution but increases the computational requirements and number of parameters substantially. Since training effectively on large datasets is computationally expensive, reducing patch size beyond a certain point is not helpful for accuracy. In the original paper, the value of p was chosen to be $n/16$, which resulted in 16×16 "words."

These patches are then unfurled into a sequence of length n^2/p^2, where the (i, j)th tile is the $k = [(i-1) * (n/p) + j]$th element in the sequence (see Figure 12.7). Each patch contains $p \times p \times 3 = 3p^2$ pixels, and the kth patch is flattened into a vector $\overline{x}_k$ of length $3p^2$. Then, an embedding (projection) matrix E [learned via backpropagation] is used to create the smaller vector $E\overline{x}_k$ for each k (much like the text transformer maps

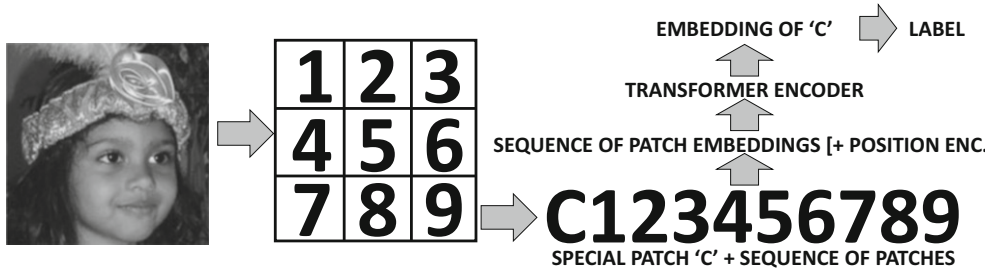


Figure 12.7: The vision transformer

one-hot encodings to word embeddings). Positional encodings are added to each $E\overline{x}_k$ based on sinusoidal functions of the sequential patch position $k = [(i-1) * (n/p) + j]$ (like the vanilla transformer). This vector sequence is pre-pended with a special patch embedding (marked by 'C' in Figure 12.7) just as BERT prepends a [CLS] token to the beginning of each sentence. The output embedding of this special patch (after applying the transformer) feeds into the output (classification) layer to yield class predictions. This simple approach treats each image as a "sentence" of n^2/p^2 words, and it is able to achieve state-of-the-art accuracy in comparison with the best convolutional neural networks. Like a vanilla transformer, the approach has the advantage of being highly parallelizeable and scalable. The fact that an image can be treated as a sequence in this way is very surprising and useful because it enables the use of off-the-shelf architectures with very few changes. Even the simple 1-dimensional sinusoidal positional encodings of the vanilla transformer worked accurately — incorporating more detailed positional encodings with both coordinates of the 2-dimensional patch position did not seem to help accuracy.

12.2.8 Attention Mechanisms in Graphs

A key difference between image convolutional networks and graph convolutional networks is that the former weights different pixels differently by multiplying with filter weights *before* aggregation. On the other hand, graph convolutional networks aggregate all neighbors before multiplication with the weight matrix. Graph attention networks weight different nodes differently in convolutional operations with the use of an attention mechanism. The convolution operation described in Equation 10.4 of Chapter 10 is repeated here:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(W^{(k)} \sum_{j \in A(i) \cup \{i\}} \frac{\overline{h}_j^{(k-1)}}{\sqrt{|A(i)| \cdot |A(j)|}}\right) \tag{12.11}$$

The notations here are the same as those in Equation 10.4 of Chapter 10, and we advise the reader to revisit that section before proceeding further. The graph is assumed to contain n vertices. The set $A(i)$ is the set of incident vertices on vertex i and $\overline{h}_i^{(k)}$ contains the p_k-dimensional activations for the ith vertex in the kth layer. The matrix of parameters $W^{(k)}$ is a $p_k \times p_{k-1}$ matrix between the $(k-1)$th layer and kth layer. The above model hard-codes the weights of activation flows between two vertices using the inverse of the geometric mean

of their degrees. In the attention-centric model, the *hard-coded degree weighting is replaced with an attention-centric weight* $\alpha_{ij}^{(k)}$, which is *learnable* in a data-driven manner:

$$(\forall i, k): \quad \overline{h}_i^{(k)} = \Phi\left(W^{(k)} \sum_{j \in A(i) \cup \{i\}} \alpha_{ij}^{(k)} \overline{h}_j^{(k-1)}\right) \tag{12.12}$$

How does one efficiently learn the *edge-specific* weights $\alpha_{ij}^{(k)}$ without overfitting, especially since the number of such weights scales with the number of edges? As is common in such settings, the parameters $\alpha_{ij}^{(k)}$ are defined indirectly in terms of a compressed attention vector $\overline{a}^{(k)}$, which is a learned $(2p_k)$-dimensional column vector. Specifically, the attention coefficients $\alpha_{ij}^{(k)}$ are defined in terms of the $(2p_k)$-dimensional attention vector $\overline{a}^{(k)}$ as follows:

$$(\forall i, j, k): \quad \alpha_{ij}^{(k)} = \text{Softmax}\left(\text{ReLU}\left(\left[\overline{a}^{(k)}\right]^T \begin{bmatrix} W^{(k)} \overline{h}_i^{(k-1)} \\ W^{(k)} \overline{h}_j^{(k-1)} \end{bmatrix}\right)\right)$$

Although we have used the ReLU activation function above for simplicity, it is often replaced by the Leaky ReLU to avoid too many nodes with zero activations. Note that the use of the softmax function is ubiquitous in attention mechanisms — here, it is applied to each vector of $\alpha_{ij}^{(k)}$ values *for fixed* i, k *and varying* j so that $\sum_j \alpha_{ij}^{(k)} = 1$ is satisfied for any particular combination of i and k. The compact attention vector $\overline{a}^{(k)}$ is learned by the backpropagation algorithm. The overall effect of the attention vector is to learn the importance of different adjacent nodes, so that activation flow is performed in a more discriminative manner with relevant vertices.

12.3 Neural Turing Machines

Neural Turing machines are neural networks with *external memory*. The basic principle of accessing memory selectively is similar to an attention mechanism. With the exception of LSTMs, most neural networks do not have the concept of persistent memory over long time scales. In fact, the notions of computation and memory are not clearly separated in traditional neural networks (including LSTMs). This is because the computations in a neural network are tightly integrated with the values in the hidden states, which serve the role of storing the intermediate results of the computations. This separation is much cleaner in a neural Turing machine. The base neural network can read or write to the external memory and therefore plays the role of a controller in guiding the computation. The ability to manipulate persistent memory, when combined with a clear separation of memory from computations, leads to a *programmable computer* that can simulate algorithms from examples of the input and output.

Neural Turing machines clearly distinguish between the external memory and the hidden states within the neural network. The hidden states within the neural network can be viewed in a similar way to CPU registers that are used for transitory computation, whereas the external memory is used for persistent computation. The external memory provides the neural Turing machine to perform computations in a similar way to how human programmers manipulate data on modern computers. This property often gives neural Turing machines much better generalizability in the learning process, as compared to LSTMs. This approach also provides a path to defining persistent data structures that are well separated from neural computations.

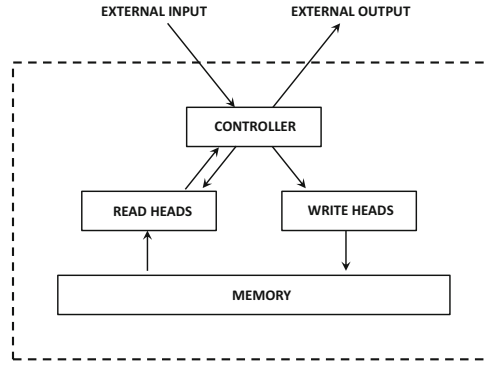


Figure 12.8: The neural Turing machine

The architecture of a neural Turing machine is shown in Figure 12.8. At the heart of the neural Turing machine is a controller, which is implemented using a recurrent neural network (although simpler alternatives are possible). The recurrent architecture carries over the state of "execution" from one time-step to the next, as the neural Turing machine implements any particular algorithm. It has the ability to exchange inputs and outputs with the environment, just like any computer program. Furthermore, it has an external memory to which it can read and write with the use of reading and writing *heads*. The memory is structured as an $N \times m$ matrix in which there are N memory cells, each of which is of length m. At the tth time-stamp, the m-dimensional vector in the ith row of the memory is denoted by $\overline{M}_t(i)$.

The heads output a *weight* $w_t(i) \in (0, 1)$ associated with each location i at time-stamp t controlling the degree to which it reads from and writes to each memory location, where the weights over all memory locations add up to 1. The weight notation uses the time-stamp t as a subscript as it changes with time. Although we have used a common notation $w_t(i)$, the weights emitted by read heads are used for reads and the weights emitted by write heads are used for writes. For example, if the read head outputs a weight of 0.1, then it interprets anything read from the ith memory location after scaling it with 0.1 and adds up the weighted reads over different values of i. Therefore, the m-dimensional vector at location i can be read as a weighted combination of the vectors in different memory locations:

$$r_t = \sum_{i=1}^{N} w_t(i)\overline{M}_t(i) \tag{12.13}$$

Since the weights over all memory locations add up to 1 (i.e., $\sum_{i=1}^{N} w_t(i) = 1$) the read value r_t is a convex combination of memory contents. It is sometimes helpful to view these weights as probabilities of reading from or writing to various memory locations, except that the operations are executed by soft weightings instead (which are differentiable operations).

Memory locations are written by first erasing a location and then adding to it. In the tth time-stamp, the write head emits a weighting vector $w_t(i)$ together with length-m erase- and add-vectors $\overline{e}_t$ and $\overline{a}_t$, respectively. Then, the update to memory location i as follows:

$$\overline{M}'_t(i) \Leftarrow \underbrace{\overline{M}_{t-1}(i) \odot (1 - w_t(i)\overline{e}_t(i))}_{\text{Soft Erase}}, \qquad \overline{M}_t(i) = \underbrace{\overline{M}'_t(i) + w_t(i)\overline{a}_t}_{\text{Soft Add}}$$

Here, the $\odot$ symbol indicates elementwise multiplication across the m dimensions of the ith row of the memory matrix. Each element in the erase vector $\overline{e}_t$ is drawn from $(0, 1)$. The m-dimensional erase vector gives fine-grained control to the choice of the elements from the m-dimensional row that can be erased. It is also possible to have multiple write heads make sequential updates, and the order of updates among the heads is immaterial because multiplications and additions are commutative and associative. However, all erases must be done before all additions to ensure a consistent result.

One can view the above updates as having an intuitively similar effect as stochastically picking one of the N rows of the memory (with probability $w_t(i)$) and then sampling individual elements (with probabilities $\overline{e}_t$) to change them. However, stochastic updates are not differentiable, whereas the soft updates of the neural Turing machine are differentiable. One can also view these weights in an analogous way to the gating mechanisms of the LSTM to regulate the amount read or written into each long-term memory location (cf. Chapter 8).

Weightings as Addressing Mechanisms

The weightings can be viewed in a similar way to how addressing mechanisms work. For example, one might have chosen to sample the ith row of the memory matrix with probability $w_t(i)$ to read or write it, which is a *hard* mechanism. The soft addressing mechanism of the neural Turing machine changes all cells by amounts proportional to $w_t(i)$. The weight $w_t(i)$ can be computed either by content or by location.

In the case of addressing by content, we denote the weight b $w_t^c(i)$ with a superscript 'c' added to the notation. A vector $\overline{v}_t$ of length-m, which is the *key vector*, is used to weight locations based on their softmax-normalized cosine similarity to the content of the memory locations $\overline{M}_t(i)$ (as in attention mechanisms):

$$w_t^c(i) = \frac{\exp(\beta_t \text{cosine}(\overline{v}_t, \overline{M}_t(i)))}{\sum_{j=1}^{N} \exp(\beta_t \text{cosine}(\overline{v}_t \cdot \overline{M}_t(j)))} \tag{12.14}$$

The *temperature parameter* $\beta_t \geq 1$ regulates the sharpness of the weights. Increasing β_t makes the approach more like hard addressing, because the weights become closer to 0 or 1 values.

The above method for choosing weights results in pure content-based addressing, which is almost like random access. For example, if the content of a memory location $\overline{M}_t(i)$ includes its location, then a key-based retrieval is like soft random access of memory. A second method of addressing is by using sequential addressing with respect to the location in the previous time-stamp and with respect to contiguous memory locations. This approach is referred to as location-based addressing. In location-based addressing, a convex combination of the weight $w_{t-1}(i)$ in the previous iteration and the content weight $w_t^c(i)$ in the current iteration is generated by an *interpolation* operation. Then, a *shifting* operation adds an element of access to contiguous (and possibly related) memory locations. Finally, the softness of the addressing is *sharpened* closer to binary values. The steps are summarized as follows:

$$\text{Content-Weights}(\overline{v}_t, \beta_t) \Rightarrow \text{Interpolation}(g_t) \Rightarrow \text{Shift}(\overline{s}_t) \Rightarrow \text{Sharpen}(\gamma_t)$$

The input parameters are shown along with each operation and are obtained from the controller. Since the generation of the content weights $w_t^c(i)$ has already been discussed, we explain the other three steps:

1. *Interpolation:* The vector $w_t(i)$ from the previous iteration is combined with the content weights $w_t^c(i)$ in the current iteration using a single interpolation weight $g_t \in (0, 1)$ that is output by the controller:

$$w_t^g(i) = g_t \cdot w_t^c(i) + (1 - g_t) \cdot w_{t-1}(i) \tag{12.15}$$

   Note that if g_t is 0, then the content is not used at all.

2. *Shift:* In this case, a rotational shift is performed in which a normalized vector over integer shifts is used. For example, consider a situation where $s_t[-1] = 0.2$, $s_t[0] = 0.5$ and $s_t[1] = 0.3$. This means that the weights should shift by -1 with gating weight 0.2, and by 1 with gating weight 0.3. The shifted vector $w_t^s(i)$ is defined as follows:

$$w_t^s(i) = \sum_{j=1}^{N} w_t^g(j) \cdot s_t[i - j] \tag{12.16}$$

   The index computation $[i-j]$ is suitably modified to allow rotational shifts by treating memory locations at 1 and N as adjacent.

3. *Sharpening:* Shifting blurs the weights by averaging them over multiple values. Therefore, sharpening biases the weights closer to 0 or 1 values without changing their ordering. A parameter $\gamma_t \geq 1$ regulates the degree of sharpening:

$$w_t(i) = \frac{[w_t^s(i)]^{\gamma_t}}{\sum_{j=1}^{N} [w_t^s(j)]^{\gamma_t}} \tag{12.17}$$

   Larger values of γ_t cause greater sharpening. The parameter γ_t plays a similar role as the temperature parameter β_t in content-based weight sharpening.

Using a gating weight g_t of 1 results in pure content-based addressing (i.e., random access), whereas values of $g_t < 1$ give non-zero weight to $w_{t-1}(i)$, enabling sequential access from the reference point of the previous step.

Architecture of Controller

An important design choice is that of the choice of the neural architecture in the controller. A natural choice is to use a recurrent neural network in which there is already a notion of temporal states. Furthermore, using an LSTM provides additional internal memory to the external memory in the neural Turing machine. The states within the neural network are like CPU registers that are used for internal computation, but they are not persistent (unlike the external memory). It is noteworthy that once we have a concept of external memory, it is not absolutely essential to use a recurrent network. This is because the memory can capture the notion of states; reading and writing from the same set of locations over successive time-stamps achieves temporal statefulness, as in a recurrent neural network. Therefore, it is also possible to use a feed-forward neural network for the controller, which provides a simpler solution.

Comparisons with LSTMs

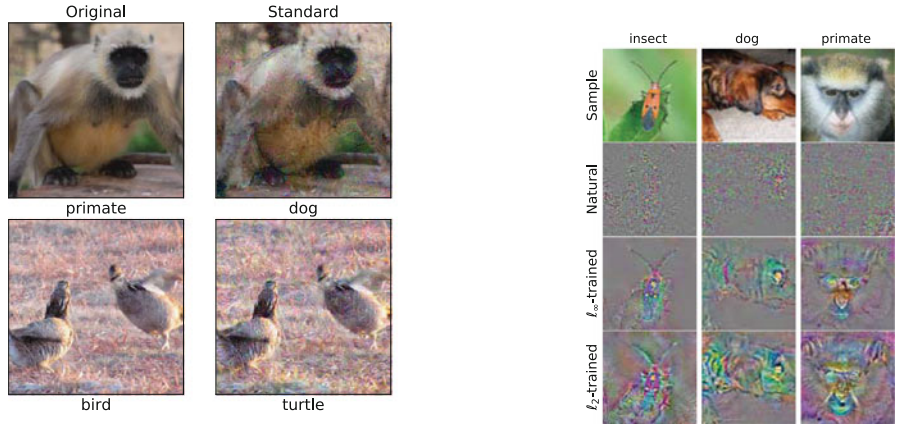
All recurrent neural networks are known to be *Turing complete* [464], which means that they can be used to simulate any algorithm. Therefore, neural Turing machines do not

theoretically add to the inherent capabilities of any recurrent neural network (including an LSTM). An experimental comparison in [165] showed that the neural Turing machine works better with much longer sequences of inputs as compared to the LSTM. One of these experiments provided both the LSTM and the neural Turing machine with pairs of input/output sequences that were identical. The goal was to copy the input to the output. In this case, the neural Turing machine generally performed better as compared to the LSTM, especially when the inputs were long. The copying algorithm of the neural Turing machine could generalize even to longer sequences than were seen during training time. However, the LSTM could perform well only on sequences of the same size range as those in the training data. In a sense, the intuitive update mechanism of the neural Turing machine provides it with helpful regularization and better generalization ability.

12.4 Adversarial Deep Learning

Since machine learning algorithms are often used to make high-stakes decisions in many real-world settings, an inevitable outcome of this increasing importance is the temptation for an adversary to try to influence the outputs of such algorithms in a desired way for motives of profit or malice. A typical approach used by an adversary is to manipulate their inputs in an imperceptible way (at prediction time), so that the outputs of the algorithm are changed as well. The key insight used by an adversary is that machine learning algorithms often generalize their models to unknown parts of the input space, since it is impossible to train a model for every possible input. Therefore, a correct output is not possible for every possible input example, and some well-chosen examples give very poor results. Although such examples might be rare *in terms of percentage-wise averages* when sampling points along random directions near training points, they are plentifully present if one chooses to dig them out by moving along specific directions near training points. The results are quite startling because it reveals that most deep learning algorithms can give surprisingly poor predictions when *imperceptibly small* changes are made to the inputs. For example, on adding some *carefully chosen* noise to the original images with true labels *primate* and *bird* in Figure 12.9(a) they are incorrectly classified as *dog* and *turtle*, respectively, even though the added noise does not make the image look any different to a human. The point is that even though deep learning might have exceeded human-level recognition in many domains like image classification, *it is surprisingly easy to make carefully calibrated changes to the input to deceive the deep learning algorithm in a way that is atypical of human vision.* Adversarial examples are specifically designed to elicit *worst-case* performance rather than *average-case* performance — they are rare in the relative sense but plentiful in the absolute sense.

While the field of adversarial machine learning is quite broad, we focus on the common approaches used to attack *deep learning* algorithms (or, more generally, machine learning algorithms based on *gradient descent*). We consider two scenarios for adversarial learning. The first is the *white-box attack* in which an adversary has access to the model used by the deep learning algorithm, and the second is the *black-box attack* in which an adversary does not have access to this model but can only observe outputs made by the deep learning algorithm. The black-box attack is more realistic, but it builds on the ideas used in white-box attacks.



(a) Examples of adversarial misclassification

(b) Adversarial training improves semantic interpretability of gradients

Figure 12.9: Examples of adversarial misclassifications and gradient-based interpretation. These figures appear in [518] (©Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, Aleksander Madry)

White-Box Attack

Consider a neural network in which the source code of the model is available to the adversary, who can therefore, calculate the gradient of the loss $L(\overline{x}, y, \overline{w})$ with respect to any particular input $\overline{x} = [x_1, \ldots, x_d]^T$ and class label y using neural network parameters $\overline{w}$. For example, each x_i might be the value of a pixel in an image, y might be the image category, and the weights $\overline{w}$ contain the neural network parameters. The adversarial attack problem can be thought of as adding a small change $\delta\overline{x}$ to $\overline{x}$ with an L_p-norm bounded above by ϵ (to keep perturbations imperceptible), so that the output loss $L(\overline{x}, y, \overline{w})$ increases as much as possible (presumably due to misclassification):

$$\delta\overline{x} = \text{argmax}_{\overline{r}}\{L(\overline{x} + \overline{r}, y, \overline{w}) : \|\overline{r}\|_p \leq \epsilon\} \tag{12.18}$$

Therefore, for a neural network with fixed weights $\overline{w}$, one can use the node-to-node derivatives (cf. section 2.3.1) in order to compute the sensitivity of the loss to the input. A particularly popular method to increase the loss is to use the *fast-gradient sign method* [151] for adding the entire perturbation in a single step with $\|\delta\overline{x}\|_\infty \leq \epsilon$:

$$\overline{x} \Leftarrow \overline{x} + \epsilon \cdot \text{sign}\left\{\frac{\partial L(\overline{x}, y, \overline{w})}{\partial \overline{x}}\right\} \tag{12.19}$$

Here, the sign function is applied element-wise to the vector of derivatives. Note that this is a gradient *ascent* update rather than a gradient descent update (since we are trying to maximize the loss), and it is executed with respect to inputs rather than weights. This is essentially a *single* projected gradient descent step that keeps the step size at the corner of a ϵ-bounding box centered at $\overline{x}$ and extending to a distance $\pm\epsilon$ in each direction. A more powerful adversarial attack explicitly uses *multi-step projected gradient descent* in which a smaller step-size $\alpha \leq \epsilon$ is used, but it is repeated multiple times and the perturbed example is projected back to the closest point on the ϵ-bounding box after each step, if the perturbed

example ever moves outside the box centered at $\overline{x}$. The multi-step version (i.e., projected gradient descent) is more powerful, but it is also more expensive. While we have used the L_∞ norm in the above example, it is also common to use L_1- and L_2-norms for the perturbation.

In some cases, the adversary may wish the neural network to produce a *desired* output such as the target class y_{false}. Therefore, the optimization problem becomes the following:

$$\delta\overline{x} = \text{argmax}_{\overline{r}}\{L(\overline{x}+\overline{r}, y, \overline{w}) - L(\overline{x}+\overline{r}, y_{false}, \overline{w}) : \|\overline{r}\|_p \leq \epsilon\}$$

A simple approach is to modify the fast gradient sign method as follows:

$$\overline{x} \Leftarrow \overline{x} + \epsilon \cdot \text{sign}\left\{\frac{\partial L(\overline{x}, y, \overline{w})}{\partial \overline{x}} - \frac{\partial L(\overline{x}, y_{false}, \overline{w})}{\partial \overline{x}}\right\} \tag{12.20}$$

It is also possible to simply minimize the loss with respect to y_{false} in order to obtain the perturbation. One surprising observation [151] is that providing the perturbed inputs from one trained model often provides similar prediction results on an independently trained model (on similar training data) but with a completely different neural architecture. The most likely reason for this is that even independent models of high capacity often inherit similar biases as those in the lowest capacity models (e.g., linear models), and therefore tend to provide similar results. This property, referred to as the *transferability of adversarial samples*, is useful for an adversary, as it can be further improved upon to create a black-box attack.

Black-Box Attack

The white-box attack is unrealistic, as adversaries rarely have access to the trained model. It is more realistic to consider cases in which an adversary can observe outputs of the trained model but not have access to the model itself. Therefore, the adversary repeatedly feeds samples to the model to observe the outputs and then records the input-output pairs to create a new training data set. Then, the adversary builds a surrogate model using the newly constructed training data set. This surrogate model is subjected to white-box attacks in order to create the perturbed input samples [376]. The principle of adversarial transferability is useful in ensuring that this arrack also works on the original model.

The main challenge in building the surrogate model is to query the original model for a sufficient number of useful inputs. Using random input samples is not advisable, as it is not sufficiently reflective of the true distribution of input data points. Ideally, we want to start with an initial (seed) set of data points that are similar to those on which the original model was trained. It is usually possible to obtain such a sample of points. Subsequently, we want to sample further points in order to crate an new model of the decision boundary that is similar to the one constructed by the original model. Obtaining training points near the decision boundary is particularly helpful in this regard. The additional training points are sampled in the neighborhood of the earlier training points by moving along directions (starting with a given input training point) where there is the maximum variation in output along the direction. The idea is that there is maximum uncertainty along that direction (which might possibly be close to a decision boundary with high variability in class distribution). Therefore, sampling along that direction helps us mimic the original distribution more closely. Let $\overline{o}$ be the output (column) vector for a given input (column) vector $\overline{x}$. Then, the Jacobian of the output with respect to the input is used to identify the

direction along which the maximum class variation is observed, and multiplied with a small step value λ:

$$\Delta\overline{x} = \lambda \cdot \text{sign}\left(\left[\frac{\partial \overline{o}}{\partial \overline{x}}\right]^T \overline{x}\right)$$

The above expression uses the denominator layout of matrix calculus, and therefore a transposition is required on the vectored derivative in order to obtain the Jacobian. This direction is used to construct the next training point $\overline{x} + \Delta\overline{x}$. This procedure is applied to each training example in the seed set, and then the procedure is repeated iteratively with the newly generated set of examples. Repeating the process to generate a large training data set helps build a good mimic neural network of the original model. The procedure is referred to as *Jacobian-based dataset augmentation*. Note that the Jacobian can be obtained using node-to-node derivatives (cf. section 2.3.1) from the backpropagation algorithm.

Adversarially Robust Training

There are significant challenges in mounting defenses against adversarial attacks, especially since the model used by the adversary is not known. Therefore, all defenses are limited in that they protect the model only in restricted situations. Another popular strategy for defending against attacks is *defensive distillation* [377]. In this case, the idea is to build a second mimic model of the original model (using an identical neural architecture), except that the *softmax outputs of the original model are used as the training samples*. This is similar to the mimic model discussed on page 159, except that the purpose here is to create a more robust (rather than compressed) model. By using the softmax outputs, we are using the probabilities of *each class* for the targets, and therefore more information is used, although it is obtained as an output of the first model. Such an approach results in a smoother decision boundary and washes away areas of brittle and unexplained model complexity (which represent a source of successful adversarial attacks).

Another simple approach is to generate a lot of adversarial examples and augment the data set with these examples. An equivalent approach, which is suggested in [151], is to use a modified objective function — for each training example $\overline{x}$, the loss is modified to a convex combination of the original loss and the perturbed loss:

$$L'(\overline{x}, y, \overline{w}) = \alpha L(\overline{x}, y, \overline{w}) + (1 - \alpha)L(\overline{x} + \delta\overline{x}, y, \overline{w})$$

Here, $\delta\overline{x}$ is a perturbation generated by a method such as the fast-gradient sign method, and $\alpha \in (0, 1)$ is the convex combination parameter. However, generating adversarial examples for training is not always effective, and it is particularly ineffective when single-step methods like the fast-gradient sign method are used. A more general approach to adversarial learning is to view it as the following minimax optimization problem [319]:

$$\text{Minimize}_{\overline{w}} \sum_{(\overline{x},y)} \text{Maximize}_{\overline{r}}\{L(\overline{x} + \overline{r}, y, \overline{w}) : \|\overline{r}\|_p \leq \epsilon\} \tag{12.21}$$

The astute reader can already see that the above objective function is directly obtained by adding an *outer optimization problem over neural network parameters* to the adversary's *inner optimization problem over input perturbations*. The order of optimization is important as it causes the adversary to be the "first mover." Since the inner and outer optimization problems are defined on disjoint sets of variables, this is a classical saddle point learning problem in traditional optimization. Such problems are generally hard to solve (in terms

of convergence behavior), although iterative methods of alternately executing optimization steps on the neural network parameters and perturbation values can sometimes provide good results. We refer the reader to [319, 679] on more details of solving this optimization problem and its various properties.

The minimax formulation suggests that there is an inherent trade-off between average-case optimization of the loss function (as in standard classification training) and the worst-case optimization inherent in adversarial classification training. Therefore, adversarially robust classifiers will tend to have lower average-case accuracy than their non-adversarial (standard) counterparts. However, adversarially robust classifiers often yield more semantically interpretable results. For example, the loss derivative with respect to each image can be treated as a pixel brightness value and plotted to show the portions of an image that contribute to a class. As evident from Figure 12.9(b), the standard model yields pixel gradients (generated without the specialized enhancements of Figure 9.13) of only limited interpretability, but those of the adversarially trained model give highly interpretable gradients. This suggests that adversarially trained models are inherently more appealing from a semantic perspective in spite of the loss of average-case accuracy.

12.5 Generative Adversarial Networks (GANs)

Generative adversarial networks can be considered an alternative model to the variational autoencoder (cf. section 5.10.4 of Chapter 5) for unsupervised generative modeling. Generative adversarial networks work with two neural network models in tandem. The first is a generative model that produces synthetic examples of objects (which models a real repository of examples) and the goal is for them to be sufficiently realistic that it is impossible to distinguish whether a particular object is real or synthetic. For example, if we have a repository of car images, the generative network will use the generative model to create synthetic examples of car images. The judgement of whether generated examples are sufficiently realistic is achieved by a second *discriminative* network, which is a binary classifier designed to recognize images as real or synthetic. The training of the generator and discriminator occurs iteratively and simultaneously using feedbacks from one another. Therefore, the two networks are adversaries, and training makes both adversaries better, until an equilibrium is reached between them. In a sense, one can view the generative network as a "counterfeiter" trying to produce fake notes, and the discriminative network as the "police" who is trying to catch the counterfeiter producing fake notes.

An incorrect classification by the discriminator causes its weights to be modified (as with any binary classifier), and a correct classification of a synthetic object by the discriminator causes the *generator* weights to me modified. The modification of generator weights is done to make it more difficult for the discriminative network to distinguish between real and fake samples. Therefore, the weights of both networks are modified with adversarial goals. Over time, the generative network gets better and better at producing counterfeits. Eventually, it becomes impossible for the discriminator to distinguish between real and synthetically generated objects. As we will see later, this adversarial approach to training boils down to a minimax optimization problem.

The generated objects are often useful for creating large amounts of synthetic data for machine learning algorithms, and may play a useful role in data augmentation. Furthermore, by adding context, it is possible to use this approach for generating objects with different properties. For example, the input might be a text caption, such as "*spotted cat with collar*," and the output will be a fantasy image matching the description [343, 408]. The generated

objects are sometimes also used for artistic endeavors. Recently, these methods have also found application in image-to-image translation. In image-to-image translation, the missing characteristics of an image are completed in a realistic way.

12.5.1 Training a Generative Adversarial Network

The training process of a generative adversarial network proceeds by alternately updating the parameters of the generator and the discriminator. Both the generator and discriminator are neural networks. The discriminator is a neural network with d-dimensional inputs and a single output in $(0, 1)$, which indicates the probability whether or not the d-dimensional input example is real (with a value of 1 indicating that the image is real).

The generator takes as input noise samples from a p-dimensional standard Gaussian distribution, and uses it to generate d-dimensional examples of the data. One can view the generator in an analogous way to the decoder portion of a variational autoencoder (cf. section 5.10.4 of Chapter 5), which also inputs noise samples and outputs data examples. The training process here is, however, very different. Instead of using the reconstruction error gradients for modifying weights, the success of the discriminator in recognizing a fake sample is used to modify generator weights, whereas the failure of the discriminator in correctly classifying an example of any type is used to modify discriminator weights.

Let R_m be m randomly sampled examples from the real data set, and S_m be m synthetic samples from the generator. The synthetic samples are generated by first creating a set N_m of p-dimensional noise samples $\{\overline{Z}_m \ldots \overline{Z}_m\}$, and then applying the generator to these noise samples as the input to create the data samples $S_m = \{G(\overline{Z}_1) \ldots G(\overline{Z}_m)\}$. Let the probabilistic output of the discriminator for input $\overline{X}$ be denoted by $D(\overline{X}) \in (0, 1)$. Then, the *maximization* objective function J_D for the discriminator is as follows:

$$\text{Maximize}_D\, J_D = \underbrace{\sum_{\overline{X} \in R_m} \log\left[D(\overline{X})\right]}_{m \text{ samples of real examples}} + \underbrace{\sum_{\overline{X} \in S_m} \log\left[1 - D(\overline{X})\right]}_{m \text{ samples of synthetic examples}}$$

It is easy to verify that this objective function will be maximized when real examples are correctly classified to 1 and synthetic examples are correctly classified to 0.

Next we define the objective function of the generator, whose goal is to fool the discriminator. For the generator, we do not care about the real examples, because the generator only cares about the sample it generates. The generator creates m synthetic samples, S_m, and the goal for the generator is to ensure that the discriminator misclassifies them. Therefore, the generator objective function, J_G, *minimizes* the likelihood that these samples are flagged as synthetic, which results in the following optimization problem:

$$\text{Minimize}_G\, J_G = \underbrace{\sum_{\overline{X} \in S_m} \log\left[1 - D(\overline{X})\right]}_{m \text{ samples of synthetic examples}} = \sum_{\overline{Z} \in N_m} \log\left[1 - D(G(\overline{Z}))\right]$$

This objective function is minimized when the synthetic examples are incorrectly classified to 1. Since the optimization variables for the generator and discriminator are disjoint, one can consolidate the two different optimization problems into a single minimax problem:

$$\text{Minimize}_G\, \text{Maximize}_D\, J_D \tag{12.22}$$

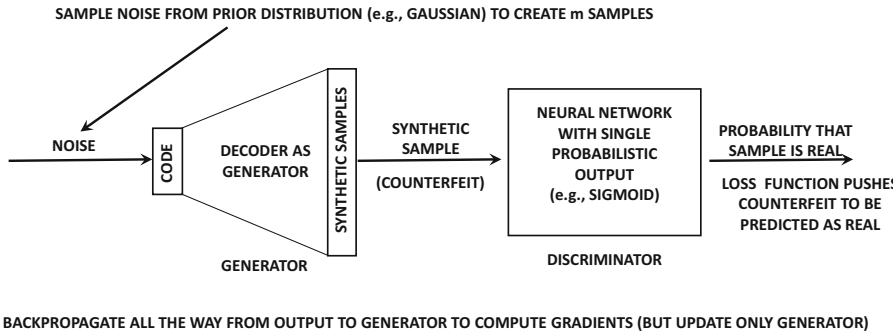


Figure 12.10: Hooked up configuration of generator and discriminator for performing gradient-descent updates on generator

The solution to such a minimax optimization problem is a *saddle point*, which is also a *Nash equilibrium* of the minimax game. An example of what a multivariate saddle point looks like is shown in Figure 4.13(b) of Chapter 4; a saddle point is a maximum for some optimization variables (discriminator parameters), whereas it is a minimum for others (generator parameters).

Stochastic gradient ascent is applied to the parameters of the discriminator and stochastic gradient descent is applied to the parameters of the generator. A single update of the generator is interleaved with k updates of the discriminator as follows:

1. **(Repeat k times):** A mini-batch of size $2 \cdot m$ is constructed with an equal number of real and synthetic examples. The synthetic examples are created by inputting noise samples to the generator from the prior distribution, whereas the real samples are selected from the base data set. Backpropagation-based Stochastic gradient ascent is performed on the parameters of the discriminator (with respect to this mini-batch) so as the maximize the likelihood that the discriminator correctly classifies both the real and synthetic examples.

2. **(Perform once):** Hook up the discriminator at the end of the generator as shown in Figure 12.10. Provide the generator with m noise inputs so as to create m synthetic examples (which is the current mini-batch). Perform stochastic gradient descent on the parameters of the generator so as to minimize the likelihood that the discriminator correctly classifies the synthetic examples. The minimization of $\log\left[1 - D(\overline{X})\right]$ in the loss function explicitly encourages these counterfeits to be predicted as real. Even though the discriminator is hooked up to the generator (and backpropagation will compute gradient updates for both), the actual updates are performed only with respect to the parameters of the generator.

The updates are repeated to convergence. The value of k is typically less than 5. If the generator is updated more frequently than the discriminator, it can readjust itself to a particular discriminator model and cause the generator to repeatedly produce very similar samples with little diversity. This phenomenon is referred to as *mode collapse* and it is a common problem with the generative adversarial network.

A heuristic adjustment to the objective function of the generator in the early iterations is to maximize $\log\left[D(\overline{X})\right]$ for each $\overline{X} \in S_m$ instead of minimizing $\log\left[1 - D(\overline{X})\right]$. The reason is that the generator will produce poor samples in early iterations and therefore $D(\overline{X})$ will

be close to 0 for all synthetic samples. As a result, the loss function of the generator will be close to 0, and its gradient will be quite modest as well. This type is saturation causes slow training of the generator parameters. Changing the optimization model to maximize $\log\left[D(\overline{X})\right]$ for the generator during early iterations improves the speed of optimization.

12.5.2 Comparison with Variational Autoencoder

The variational autoencoder and the generative adversarial network were developed independently at around the same time, and the two models accomplish similar goals. Unlike a variational autoencoder, only a decoder (i.e., generator) is learned, and an encoder is not learned by the generative adversarial network. Therefore, a generative adversarial network is not designed to reconstruct specific input samples like a variational autoencoder. However, both models can generate images like the base data, because the hidden space has a known structure (typically Gaussian) from which points can be sampled. In general, the generative adversarial network produces samples of better quality (e.g., less blurry images) than a variational autoencoder. This is because the adversarial approach is specifically designed to produce realistic images, whereas the regularization of the variational autoencoder actually hurts the quality of the generated objects. Furthermore, when reconstruction error is used to create an output for a specific image in the variational autoencoder, it forces the model to average over all plausible outputs. Averaging over plausible outputs, which are often slightly shifted from one another, is a direct cause of blurriness. On the other hand, the generative adversarial network is specifically designed to produce objects of a quality that fool the discriminator, which causes the generated images to be sharp and realistic.

The training principles of the two models are very different. The variational autoencoder *directly* sees the real images in the training process, whereas the generator of the generative adversarial network never sees the real images directly — only the discriminator parameters are updated for real images. The generator of the adversarial network is updated only when the discriminator correctly classifies a synthetic example. Therefore, the feedback from real images to the generator is very indirect and it passes through the judgements of the discriminator model (which gives images their realistic quality).

12.5.3 Using GANs for Generating Image Data

GAN is commonly used is for generating image objects with varying types of context. Indeed, the image setting is, by far, the most common use case of GANs. The generator for the image setting is referred to as a *deconvolutional network*. The most popular way to design a deconvolutional network for the GAN is discussed in [398]. Therefore, the corresponding GAN is also referred to as a DCGAN. It is noteworthy that the term "deconvolution" has generally been replaced by transposed convolution in recent years, because the former term is somewhat misleading.

The work in [398] starts with 100-dimensional Gaussian noise, which is the starting point of the decoder. This 100-dimensional Gaussian noise is reshaped into 1024 feature maps of size 4×4. This is achieved with a fully connected matrix multiplication with the 100-dimensional input, and the result is reshaped into a tensor. Subsequently, the depth of each layer reduces by a factor of 2, while increasing the lengths and widths by a factor of 2. For example, the second layer contains 512 feature maps, whereas the third layer contains 256 feature maps.

However, increasing length and width with convolution seems odd, because a convolution with even a stride of 1 tends to reduce spatial map size (unless one uses additional zero

padding). So how can one use convolutions to increase lengths and widths by a factor of 2? This is achieved by using *fractionally strided convolutions* or *transposed convolutions* at a fractional value of 0.5. These types of transposed convolutions are described at the end of section 9.5.2 of Chapter 9. The case of fractional strides is not very different from unit strides, and it can be conceptually viewed as a convolution performed after stretching the input volume spatially by either inserting zeros between rows/columns or by inserted interpolated values. Since the input volume is already stretched by a particular factor, applying convolution with stride 1 on this input is equivalent to using fractional strides on the original input. An alternative to the approach of fractionally strided convolutions is to use pooling and unpooling in order to manipulate the spatial footprints. When fractionally strided convolutions are used, no pooling or unpooling needs to be used. An overview of the architecture of the generator in DCGAN is given in Figure 12.11. A detailed discussion of the convolution arithmetic required for fractionally strided convolutions is available in [111].

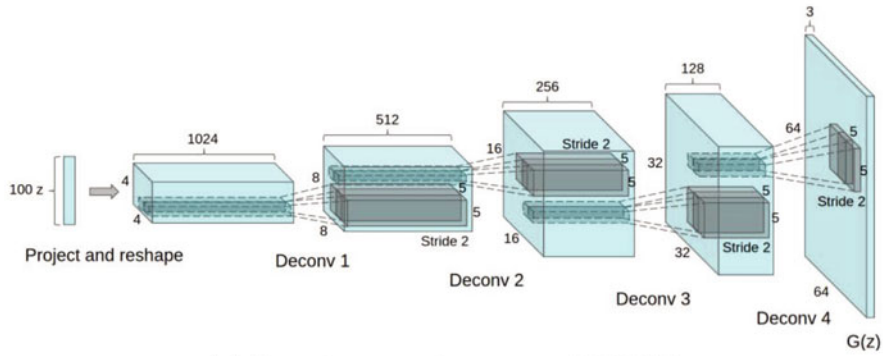
The generated images are sensitive to the noise samples. Figure 12.11(b) shows examples of the images are generated using the different noise samples. An interesting example is shown in the sixth row in which a room without a window is gradually transformed into one with a large window [398]. Such smooth transitions are also observed in the case of the variational autoencoder. The noise samples are also amenable to vector arithmetic, which is semantically interpretable. For example, one would subtract a noise sample of a neutral woman from that of a smiling woman and add the noise sample of a smiling man. This noise sample is input to the generator in order to obtain an image sample of a smiling man. This example [398] is shown in Figure 12.11(c).

The discriminator also uses a convolutional neural network architecture, except that the leaky ReLU was used instead of the ReLU. The final convolutional layer of the discriminator is flattened and fed into a single sigmoid output. Fully connected layers were not used in either the generator or the discriminator. As is common in convolutional neural networks, the ReLU activation is used. Batch normalization was used in order to reduce any problems with the vanishing and exploding gradient problems [225].

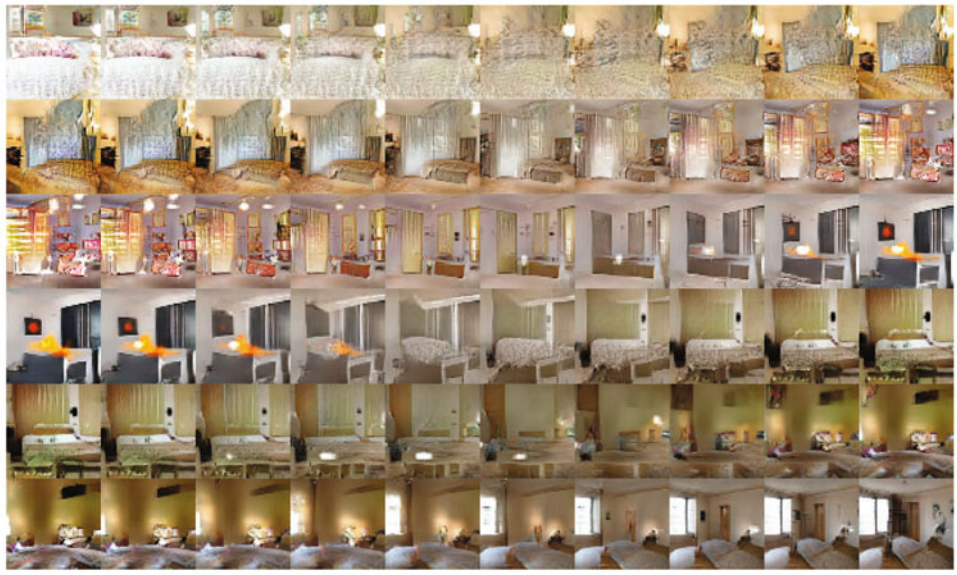
12.5.4 Conditional Generative Adversarial Networks

In conditional adversarial generative networks (CGANs), both the generator and the discriminator are conditioned on an additional input object, which might be a label, a caption, or even another object of the same type. In this case, the input typically correspond to *associated pairs of target objects and contexts*. The contexts are typically related to the target objects in some domain-specific way, which is learned by the model. For example, a context such as "*smiling girl*" might provide an image of a smiling girl. Here, it is important to note that there are many possible choices of images that the CGAN can create for smiling girls, and the specific choice depends on the value of the noise input. Therefore, the CGAN can create a universe of target objects, based on the specific conditioning.

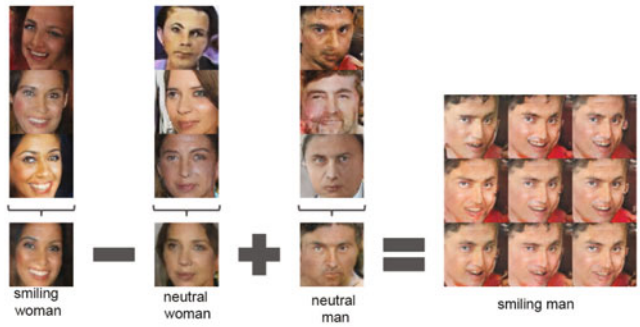
Examples of different types of conditioning in conditional GANs are shown in Figure 12.12. The context provides the additional input needed for the conditioning. In general, the context may be of any object data type, and the generated output may be of any other data type. The more interesting cases of CGAN use are those in which the context contains much less complexity (e.g., a caption) as compared to the generated output (e.g., image). In such cases, CGANs show a certain level of creativity in filling in missing details. These details can change depending on the noise input to the generator. Some examples of the object-context pairs may be as follows:



(a) Convolution architecture of DCGAN



(b) Smooth image transitions caused by changing input noise are shown in each row



(c) Arithmetic operations on input noise have semantic significance

Figure 12.11: The convolutional architecture of DCGAN and generated images. These figures appeared in [A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434, 2015]. c 2015 Alec Radford. Used with permission.

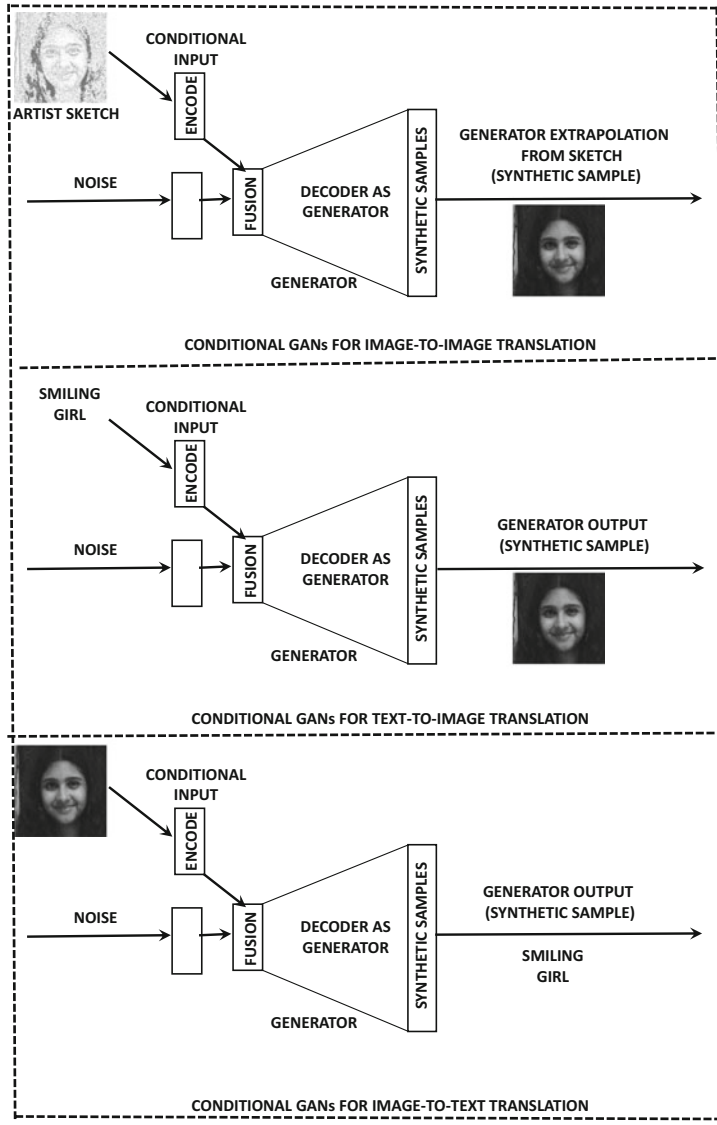


Figure 12.12: Different types of conditional generators for adversarial network. The examples are only illustrative in nature, and they do not reflect actual CGAN output.

1. Each object may be associated with a label. The label provides the conditioning for generating images. For example, in the MNIST data set (cf. Chapter 1), the conditioning might be a label value from 0 to 9, and the generator is expected to create an image of that digit, when provided that conditioning. Similarly, for an image data set, the conditioning might be a label like "*carrot*" and the output would be an image of a carrot. The experiments in the original work on conditional adversarial nets [343] generated a 784-dimensional representation of a digit based on a label from 0 to 9. The base examples of the digits were obtained from the MNIST data set (cf. section 1.7.1 of Chapter 1).

2. The target object and its context might be of the same type, although the context might be missing the rich level of detail in the target object. For example, the context might be a human artist's sketch of a purse, and the target object might be an actual photograph of the same purse with all details filled in. Another example could be an artist's sketch of a criminal suspect (which is the context), and the target object (output of generator) could be an extrapolation of the actual photograph of the person. The goal is to use a given sketch to generate various realistic samples with details filled in. Such an example is illustrated in the top part of Figure 12.12. When the contextual objects have complex representations such as images or text sentences, they may need to be converted to a multidimensional representation with an encoder, so that they can be fused with multidimensional Gaussian noise. Depending on the specific data type, the encoder might be a convolutional or recurrent neural network.

3. Each object might be associated with a textual description (e.g., image with caption), and the latter provides the context. The caption provides the conditioning for the object. The idea is that by providing a context like "*blue bird with sharp claws*," the generator should provide a fantasy image that reflects this description. An example of an illustrative image generated using the context "*smiling girl*" is illustrated in Figure 12.12. Note that it is also possible to use an image context, and generate a caption for it using a GAN, as shown in the bottom of the figure. However, it is more common to generate complex objects (e.g., images) from simpler contexts (e.g., captions) rather than the reverse. This is because a variety of more accurate supervised learning methods are available when one is trying to generate simple objects (e.g., labels or captions) from complex objects (e.g., images).

4. The base object might be a photograph or video in black and white (e.g., classic movie), and the output object might be the color version of the object. In essence, the GAN learns from examples of such pairs what is the most realistic way of coloring a black-and-white scene. For example, it will use the colors of trees in the training data to give corresponding colors in the generated object without changing its basic outline.

In all these cases, it is evident that GANs are very good at *filling in missing information*. The unconditional GAN is a special case of this setting in which all forms of context are missing, and therefore the GAN is forced to create an image without any information. The conditional case is potentially more interesting from an application-centric point of view because one often has setting where a small amount of partial information is available, and one must extrapolate in a realistic way. When the amount of available context is very small, missing data analysis methods will not work because they require significantly more context to provide reconstructions. On other hand, GANs do not promise faithful reconstructions (like autoencoders or matrix factorization methods), but they provide realistic extrapolations in which missing details are filled into the object in a realistic and harmonious way. As a result, the GAN uses this freedom to generate samples of high quality, rather than a blurred estimation of the average reconstruction. Although a given generation may not perfectly reflect a given context, one can always generate multiple samples in order to explore different types of extrapolations of the same context. For example, given the sketch of a criminal suspect, one might generate different photographs with varying details that are not present in the sketch. In this sense, generative adversarial networks exhibit a certain level of artistry/creativity that is not present in conventional data reconstruction methods. This type of creativity is essential when one is working with only a small amount of context

to begin with, and therefore the model needs to be have sufficient freedom to fill in missing details in a reasonable way.

It is noteworthy that a wide variety of machine learning problems (including classification) can be viewed as missing data imputation problems. Technically, the CGAN can be used for these problems as well. However, the CGAN is more useful for specific types of missing data, where the missing portion is too large to be faithfully reconstructed by the model. Although one can even use a CGAN for classification or image captioning, this is obviously not the best use[2] of the generator model, which is tailored towards generative creativity. When the conditioning object is more complex as compared to the output object, it is possible to get into situations where the CGAN generates a fixed output irrespective of input noise.

In the case of the generator, the inputs correspond to a point generated from the noise distribution and the conditional object, which are combined to create a single hidden code. This input is fed into the generator (decoder), which creates a conditioned sample for the data. For the discriminator, the input is a sample from the base data and its context. The base object and its conditional input are first fused into a hidden representation, and the discriminator then provides a classification of whether it is real or generated. The overall architecture for the training of the generator portion is shown in Figure 12.13. It is instructive to compare this architecture with that of the unconditional GAN in Figure 12.10. The main difference is that an additional conditional input is provided in the second case. The loss function and the overall arrangement of the hidden layers is very similar in both cases. Therefore, the change from an unconditional GAN to a conditional GAN requires only minor changes to the overall architecture. The backpropagation approach remains largely unaffected, except that there are some additional weights in the portion of the neural network associated with the conditioning inputs that one might need to update.

An important point about using GANs with various data types is that they might require some modifications in order to perform the encoding and decoding in a data-sensitive way. While we have given several examples from the image and text domain in our discussion

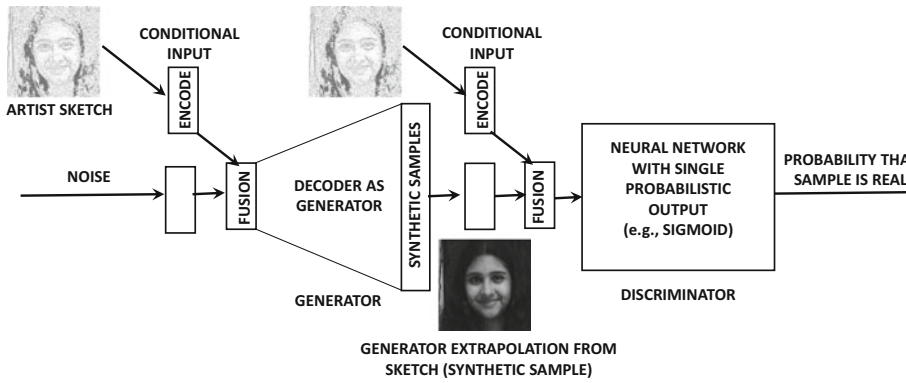


Figure 12.13: Conditional generative adversarial network for hooked up discriminator: It is instructive to compare this architecture with that of the unconditional generative adversarial network in Figure 12.10.

[2]It turns out that by modifying the *discriminator* to output classes (including the *fake* class), one can obtain state-of-the-art semi-supervised classification with very few labels [437]. However, using the *generator* to output the labels is not a good choice.

above, most of the description of the algorithm is focussed on vanilla multidimensional data (rather than image or text data). Even when the label is used as the context, it needs to be encoded into a multidimensional representation (e.g., one-hot encoding). Therefore, both Figures 12.12 and 12.13 contain a specifically denoted component for encoding the context. In the earliest work on conditional GANs [343], the pre-trained *AlexNet* convolution network [263] is used as the encoder for image context (without the final label prediction layer). *AlexNet* was pre-trained on the *ImageNet* database. The work in [343] even uses a multimodal setting in which an image is input together with some text annotations. The output is another set of text tags further describing the image. For text annotations, a pre-trained *word2vec* (skip-gram) model is used as the encoder. It is noteworthy that it is even possible to fine-tune the weights of these pre-trained encoder networks while updating the weights of the generator (by backpropagating beyond the generator into the encoder). This is particularly useful if the nature of the data set for object generation in the GAN is very different from the data sets on which the encoders were pretrained. However, the original work in [343] fixed these encoders to their pre-trained configurations, and was still able to generate reasonably high-quality results.

12.6 Competitive Learning

Most of the learning methods discussed in this book are based on updating the weights in the neural network in order to correct for errors. Competitive learning is a fundamentally different paradigm in which the goal is not to map inputs to outputs in order to correct errors. Rather, the neurons compete for the right to respond to a subset of similar input data and push their weights closer to one or more input data points. Therefore, the learning process is also very different from the backpropagation algorithm used in neural networks.

The broad idea in training is as follows. The activation of an output neuron increases with greater similarity between the weight vector of the neuron and the input. It is assumed that the weight vector of the neuron has the same dimensionality as the input. A common approach is to use the Euclidian distance between the input and the weight vector in order to compute the activation. Smaller distances lead to larger activations. The output unit that has the highest activation to a given input is declared the winner and moved closer to the input.

In the winner-take-all strategy, only the winning neuron (i.e., neurons with largest activation) is updated and the remaining neurons remain unchanged. Other variants of the competitive learning paradigm allow other neurons to participate in the update based on pre-defined neighborhood relationships. Furthermore, some mechanisms are also available that allow neurons to inhibit one another. These mechanisms are forms of regularization that can be used to learn representations with a specific type of pre-defined structure, which is useful in applications like 2-dimensional visualization. First, we discuss a simple version of the competitive learning algorithm in which the winner-take-all approach is used.

Let $\overline{X}$ be an input vector in d dimensions, and $\overline{W}_i$ be the weight vector associated with the ith neuron in the same number of dimensions. Assume that a total of m neurons is used, where m is typically much less than the size of the data set n. The following steps are used by repeatedly sampling $\overline{X}$ from the input data and making the following computations:

1. The Euclidean distance $||\overline{W}_i - \overline{X}||$ is computed for each i. If the pth neuron has the smallest value of the Euclidean distance, then it is declared as the winner. Note that the value of $||\overline{W}_i - \overline{X}||$ is treated as the activation value of the ith neuron.

2. The pth neuron is updated using the following rule:

$$\overline{W}_p \Leftarrow \overline{W}_p + \alpha(\overline{X} - \overline{W}_p) \tag{12.23}$$

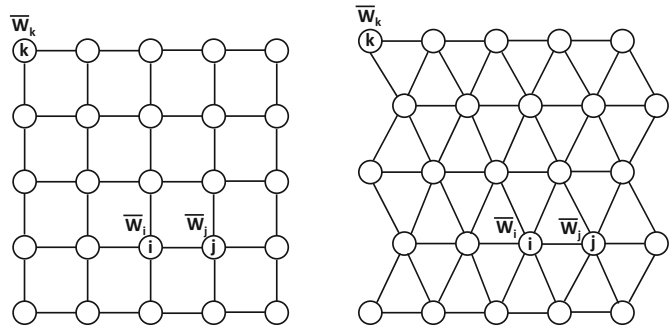
Here, $\alpha > 0$ is the learning rate. Typically, the value of α is much less than 1. In some cases, the learning rate α reduces with progression of the algorithm.

The basic idea in competitive learning is to view the weight vectors as prototypes (like the centroids in k-means clustering), and then move the (winning) prototype a small distance towards the training instance. The value of α regulates the fraction of the distance between the point and the weight vector, by which the movement of $\overline{W}_p$ occurs. Note that k-means clustering also achieves similar goals, albeit in a different way. After all, when a point is assigned to the winning centroid, it moves that centroid by a small distance towards the training instance at the end of the iteration. Competitive learning allows some natural variations of this framework, which can be used for unsupervised applications like clustering and dimensionality reduction.

12.6.1 Vector Quantization

Vector quantization is the simplest application of competitive learning. Some changes are made to the basic competitive learning paradigm with the notion of *sensitivity*. Each node has a sensitivity $s_i \geq 0$ associated with it. The sensitivity value helps in balancing the points among different clusters. The basic steps of vector quantization are similar to those in the competitive learning algorithm except for differences caused by how s_i is updated and used in the computations. The value of s_i is initialized to 0 for each point. In each iteration, the value of s_i is increased by $\gamma > 0$ for non-winners and set to 0 for the winner. Furthermore, to choose the winner, the smallest value of $||\overline{W}_i - \overline{X}|| - s_i$ is used. Such an approach tends to make the clusters more balanced, even if the different regions have widely varying density. This approach ensures that points in dense regions are typically very close to one of the weight vectors and the points in sparse regions are approximated very poorly. Such a property is common in applications like dimensionality reduction and compression. The value of γ regulates the effect of sensitivity. Setting γ to 0 reverts to pure competitive learning as discussed above.

The most common application of vector quantization is compression. In compression, each point is represented by its closest weight vector $\overline{W}_i$, where i ranges from 1 to m. Note that the value of m is much less than the number of points n in the data set. The first step is to construct a code book containing the vectors $\overline{W}_1 \ldots \overline{W}_m$, which requires a space of $m \cdot d$ for a data set of dimensionality d. Each point is stored as an index value from 1 through m, depending on its closest weight vector. However, only $\log_2(m)$ bits are required in order to store each data point. Therefore, the overall space requirement is $m \cdot d + \log_2(m)$, which is typically much less than the original space required $n \cdot d$ of the data set. For example, a data set containing 10 billion points in 100 dimensions requires space in the order of 4 Terabytes, if 4 bytes are required for each dimension. On the other hand, by quantizing with $m = 10^6$, the space required for the code-book is less than half a Gigabyte, and 20 bits are required for each point. Therefore, the space required for the points (without the code-book) is less than 3 Gigabytes. Therefore, the overall space requirement is less than 3.5 Gigabytes including the code-book. Note that this type of compression is lossy, and the error of the approximation of the point $\overline{X}$ is $||\overline{X} - \overline{W}_i||$. Points in dense regions are approximated very well, whereas outliers in sparse regions are approximated poorly.



(a) Rectangular (b) Hexagonal

Figure 12.14: An example of a 5×5 lattice structure for the self-organizing map. Since neurons i and j are close in the lattice, the learning process will bias the values of $\overline{W}_i$ and $\overline{W}_j$ to be more similar. The rectangular lattice will lead to rectangular clustered regions in the resulting 2-dimensional representation, whereas the hexagonal lattice will lead to hexagonal clustered regions in the resulting 2-dimensional representation

12.6.2 Kohonen Self-Organizing Map

The Kohonen self-organizing map is a variation on the competitive learning paradigm in which a 1-dimensional string-like or 2-dimensional lattice-like structure is imposed on the neurons. This type of lattice structure enables the mapping of all points to 1-dimensional or 2-dimensional space for visualization. An example of a 2-dimensional lattice structure of 25 neurons arranged in a 5×5 rectangular grid is shown in Figure 12.14(a). A hexagonal lattice containing the same number of neurons is shown in Figure 12.14(b). The shape of the lattice affects the shape of the 2-dimensional regions in which the clusters will be mapped. The case of 1-dimensional string-like structure is similar. The idea of using the lattice structure is that the values of $\overline{W}_i$ in adjacent lattice neurons tend to be similar. Here, it is important to define separate notations to distinguish between the distance $||\overline{W}_i - \overline{W}_j||$ and the distance on the lattice. The distance between adjacent pairs of neurons on the lattice is exactly one unit. For example, the distance between the neurons i and j based on the lattice structure in Figure 12.14(a) is 1 unit, and the distance between neurons i and k is $\sqrt{2^2 + 3^2} = \sqrt{13}$. The vector-distance in the original input space (e.g., $||\overline{X} - \overline{W}_i||$ or $||\overline{W}_i - \overline{W}_j||$) is denoted by a notation like $Dist(\overline{W}_i, \overline{W}_j)$. On the other hand, the distance between neurons i and j along the lattice structure is denoted by $LDist(i, j)$. Note that the value of $LDist(i, j)$ is dependent only on the indices (i, j), and is independent of the values of the vectors $\overline{W}_i$ and $\overline{W}_j$.

The learning process in the self-organizing map is regulated in such a way that the closeness of neurons i and j (based on lattice distance) will also bias their weight vectors to be more similar. In other words, *the lattice structure of the self-organizing maps acts as a regularizer in the learning process*. As we will see later, imposing this type of 2-dimensional structure on the learned weights is helpful for visualizing the original data points with a 2-dimensional embedding.

The overall self-organizing map training algorithm proceeds in a similar way to competitive learning by sampling $\overline{X}$ from the training data, and finding the winner neuron based on the Euclidean distance. The weights in the winner neuron are updated in a manner similar to the vanilla competitive learning algorithm. However, the main difference is that a damped version of this update is also applied to the lattice-neighbors of the winner neuron. In fact, in soft variations of this method, one can apply this update to all neurons, and the

level of damping depends on the lattice distance of that neuron to the winning neuron. The damping function, which always lies in $[0, 1]$, is typically defined by a Gaussian kernel:

$$Damp(i, j) = \exp\left(-\frac{LDist(i, j)^2}{2\sigma^2}\right) \tag{12.24}$$

Here, σ is the bandwidth of the Gaussian kernel. Using extremely small values of σ reverts to pure winner-take-all learning, whereas using larger values of σ leads to greater regularization in which lattice-adjacent units have more similar weights. For small values of σ, the damping function will be 1 only for the winner neuron, and it will be 0 for all other neurons. Therefore, the value of σ is one of the parameters available to the user for tuning. Note that many other kernel functions are possible for controlling the regularization and damping. For example, instead of the smooth Gaussian damping function, one can use a thresholded step kernel, which takes on a value of 1 when $LDist(i, j) < \sigma$, and 0, otherwise.

The training algorithm repeatedly samples $\overline{X}$ from the training data, and computes the distances of $\overline{X}$ to each weight $\overline{W}_i$. The index p of the winning neuron is computed. Rather than applying the update only to $\overline{W}_p$ (as in winner-take-all), the following update is applied to each $\overline{W}_i$:

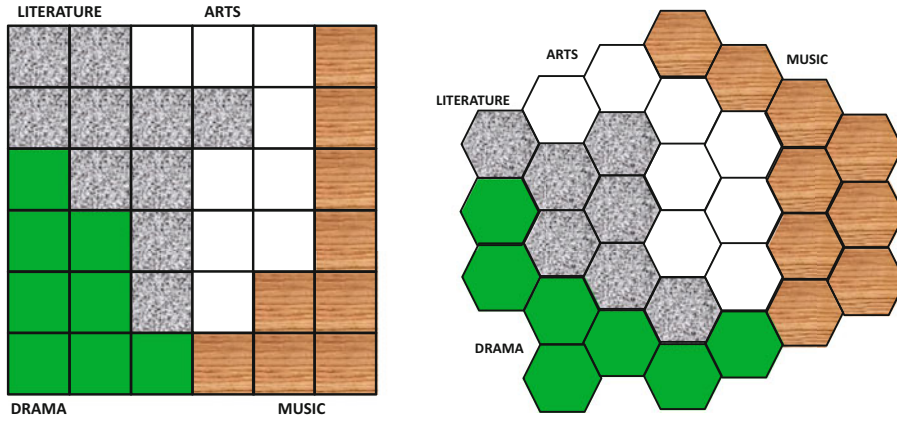
$$\overline{W}_i \Leftarrow \overline{W}_i + \alpha \cdot Damp(i, p) \cdot (\overline{X} - \overline{W}_i) \quad \forall i \tag{12.25}$$

Here, $\alpha > 0$ is the learning rate. It is common to allow the learning rate α to reduce with time. These iterations are continued until convergence is reached. Note that weights that are lattice-adjacent will receive similar updates, and will therefore tend to become more similar over time. *Therefore, the training process forces lattice-adjacent clusters to have similar points, which is useful for visualization.*

Using the Learned Map for 2D Embeddings

The self-organizing map can be used in order to induce a 2-dimensional embedding of the points. For a $k \times k$ grid, all 2-dimensional lattice coordinates will be located in a square in the positive quadrant with vertices $(0, 0)$, $(0, k-1)$, $(k-1, 0)$, and $(k-1, k-1)$. Note that each grid point in the lattice is a vertex with integer coordinates. The simplest 2-dimensional embedding is simply by representing each point $\overline{X}$ with its closest grid point (i.e., winner neuron). However, such an approach will lead to overlapping representations of points. Furthermore, a 2-dimensional representation of the data can be constructed and each coordinate is one of $k \times k$ values from $\{0 \ldots k-1\} \times \{0 \ldots k-1\}$. This is the reason that the self-organizing map is also referred to as a *discretized* dimensionality reduction method. It is possible to use various heuristics to disambiguate these overlapping points. When applied to high-dimensional document data, a visual inspection often shows documents of a particular topic being mapped to a particular local regions. Furthermore, documents of related topics (e.g., politics and elections) tend to get mapped to adjacent regions. Illustrative examples of how a self-organizing map arranges documents of four topics with rectangular and hexagonal lattices are shown in Figures 12.15(a) and (b), respectively. The regions are colored differently, depending on the majority topic of the documents belonging to the corresponding region.

Self-organizing maps have a strong neurobiological basis in terms of their relationship with how the mammalian brain is structured. In the mammalian brain, various types of sensory inputs (e.g., touch) are mapped onto a number of folded planes of cells, which are referred to as *sheets* [134]. When parts of the body that are close together receive an input (e.g., tactile input), then groups of cells that are physically close together in the brain



(a) Rectangular lattice (b) Hexagonal lattice

Figure 12.15: Examples of 2-dimensional visualization of documents belonging to four topics

will also fire together. Therefore, proximity in (sensory) inputs is mapped to proximity in neurons, as in the case of the self-organizing map. As with the neurobiological inspiration of convolutional neural networks, such insights are always used for some form of regularization.

Although Kohonen networks are used less often in the modern era of deep learning, they have significant potential in the unsupervised setting. Furthermore, the basic idea of competition can even be incorporated in multi-layer feed-forward networks. Many competitive principles are often combined with more traditional feed-forward networks. For example, the r-sparse and winner-take-all autoencoders (cf. section 5.10.1.1 of Chapter 3) are both based on competitive principles. Even the notions of attention discussed in this chapter use competitive principles in terms of focusing on a subset of the activations. Therefore, even though the self-organizing map has become less popular in recent years, the basic principles of competition can also be integrated with traditional feed-forward networks.

12.7 Limitations of Neural Networks

Deep learning has made significant progress in recent years, and has even outperformed humans on many tasks like image classification. Similarly, the success of reinforcement learning to show super-human performance in some games that require sequential planning has been quite extraordinary. Therefore, it is tempting to posit that artificial intelligence might eventually come close to or even exceed the abilities of humans in a more generic way. However, there are several fundamental technical hurdles that need to be crossed before we can build machines that learn and think like people [267]. In particular, neural networks require large amounts of training data to provide high-quality results, which is significantly inferior to human abilities. Furthermore, the amount of energy required by neural networks for various tasks far exceeds that consumed by the human for similar tasks. These observations put fundamental constraints on the abilities of neural networks to exceed certain parameters of human performance. In the following, we discuss these issues along with some recent research directions.

12.7.1 An Aspirational Goal: Few Shot Learning

Although deep learning has received increasing attention in recent years because of its success on large-scale learning tasks (compared to the mediocre performance in early years on smaller data sets), this also exposes an important weakness in current deep learning technology. For tasks like image classification, where deep learning has exceeded human performance, it has done so in a *sample-inefficient fashion*. For example, the *ImageNet* database contains more than a million images, and a neural network will often require thousands of samples of a class in order to properly classify it. Humans do not require tens of thousands of images of a truck, to learn that it is a truck. If a child is shown a truck once, she will often be able to recognize another truck even when it is of a somewhat different model, shape, and color. This suggests that humans have much better ability to generalize to new settings as compared to artificial neural networks. The general principle of being able to learn from very few examples is referred to as *few-shot learning*. Learning with few examples is also closely related to *Artificial General Intelligence*, because few shot learning is often achieved by simultaneously learning *across many tasks* in a more *general* way (like humans do).

The ability of humans to generalize with fewer examples is not surprising because the connectivity of the neurons in the human brain has evolved over millions of years with a genetic algorithm, and has been passed down from generation to generation. In an indirect sense, the human neural connection structure already encodes a kind of "knowledge" gained from the evolutionary experience over millions of years. Furthermore, humans also gain knowledge over their lifetime over a variety of tasks, which helps them learn specific tasks faster. Subsequently, learning to do *specific* tasks (like recognizing a truck) is simply a fine-tuning of the encoding a person is both born with and that which one gains over the course of a lifetime. In other words, humans are masters of transfer learning both within and across generations. One can view this long-term process as an *unsupervised learning*, and the task-specific learning as *supervised fine-tuning*. The basic idea is to either use unsupervised learning or reuse the knowledge learned in earlier tasks to learn another task.

A general form of learning that supports multiple tasks with relatively few task-specific examples to as *meta-learning* or *learning-to-learn*. Thrun and Platt defined [514] learning-to-learn as follows. Given a family of tasks, a training experience for each task, and a family of performance measures (one for each task), an algorithm is said to *learn-to-learn* if its performance at each task improves both with experience *and* the number of tasks. Therefore, the rapid learning occurs within a task, whereas the learning is guided by knowledge gained more gradually across tasks, which captures the way in which task structure varies across target domains [433]. It is this variation across tasks that makes learning-to-learn challenging. The ability of learning-to-learn is a defining biological quality, because living organisms naturally use their experience over earlier tasks to show improved performance even at weakly related tasks.

Central to the idea of knowledge reuse is the concept that some aspects of learning have broad generalizability. Loosely speaking, even the pre-training of neural networks is an example of learning-to-learn, because the features learned from training on one task can be used for another task. For example, in a convolutional neural network, the features in many of the early layers are primitive shapes (e.g., edges), and they retain their usability irrespective of the kind of task and data set that they are applied on. As discussed in Chapter 9, convolutional neural networks like *AlexNet* [263] are often pre-trained on large image repositories like *ImageNet*. Subsequently, when the neural network needs to be applied to a new data set or task, the weights can be fine-tuned with the new data set. Often, far

fewer number of examples are required for this fine-tuning, because most of the basic features learned in earlier layers do not change with the data set or task at hand. The learned features can also be generalized across tasks by removing the later layers or the network and adding additional task-specific layers. Therefore, the pre-training of convolutional neural networks already provides a basic example of the concept of knowledge reuse across tasks.

Early work on few-shot learning [121] used Bayesian frameworks in order to transfer the learned knowledge from one category to the next. Some successes have been shown at meta-learning with the use of structured architectures that leverage the notions of attention, recursion, and memory. In particular, good results have been shown on the task of learning across categories with neural Turing machines [433]. Neural Turing machines have also been used to recognize new classes with one-shot learning [527].

Meta-learning has also been used in natural language processing, where language models are seen as unsupervised multitask learners as discussed in the GPT-2 paper [399]. The GPT-3 [50] and and BERT [101] models are built on larger corpora of text. These models are then fine-tuned to specific tasks like question answering with only a small number of examples. However, the performance on some critical tasks that requite common-sense reasoning remains poor. Nevertheless, the outstanding generalizability of these models to many tasks in natural language processing has been a breakthrough that has brought us closer to Artificial General Intelligence at least in a domain-specific setting.

12.7.2 An Aspirational Goal: Energy-Efficient Learning

Deep learning models often train on powerful hardware that require large amounts of power. Using multiple GPU units in parallel typically requires power in the multi-kilowatt range. On the other hand, a human brain barely requires twenty watts to function, which is much less than the power required by a light bulb. Another point is that the human brain often does not perform detailed computations exactly, but simply makes estimates. In many learning settings, this is sufficient and can sometimes even add to generalization power. This suggests that energy-efficiency may sometimes be found in architectures that emphasize generalization over accuracy.

Several algorithms have recently been developed that trade-off accuracy in computations for improved power-efficiency of computations. Some of these methods also show improved generalization because of the noise effects of the low-precision computations. The work in [86] proposes methods for using binary weights in order to perform efficient computations. An analysis of the effect of using different representational codes on energy efficiency is provided in [295]. Certain types of neural networks, which contain *spiking neurons*, are known to be more energy-efficient [60]. The notion of spiking neurons is inspired by the biological model of the mammalian brain. The basic idea is that a neuron does not fire at each propagation cycle, but only when its *membrane potential* reaches a specific value.

Energy efficiency is often achieved when the size of the neural network is small, and redundant connections are pruned. Removing redundant connections also helps in regularization. The work in [179] proposes to learn weights and connections in neural networks simultaneously by pruning redundant connections. In particular, weights that are close to zero can be removed. As discussed in Chapter 5, training a network to give near-zero weights can be achieved with L_1-regularization. However, the work in [179] shows that L_2-regularization gives higher accuracy. Therefore, the work in [179] uses L_2-regularization and prunes the weights that are below a particular threshold. The pruning is done in an iterative fashion, where the weights are retrained after pruning them, and then the low-weight edges are pruned again. In each iteration, the trained weights from the previous phase are

used for the next phase. As a result, the dense network can be sparsified into a network with far fewer connections. Furthermore, the dead neurons that have zero input connections and output connections are pruned. Further enhancements were reported in [178], where the approach was combined with Huffman coding and quantization for compression. The goal of quantization is to reduce the number of bits representing each connection. This approach reduced the storage required by *AlexNet* [263] by a factor of 35, from about 240MB to 6.9MB with no loss of accuracy. As a result, it becomes possible to fit the model into on-chip SRAM cache rather than off-chip DRAM memory. This has advantages from the perspective of speed, energy efficiency, as well as the ability to perform mobile computation in embedded devices. A hardware accelerator has been used in [178] to achieve these goals, and this acceleration is enabled by the ability to fit the model on the SRAM cache.

Another direction is to develop hardware that is tailored directly to neural networks. It is noteworthy that there is no distinction between software and hardware in humans; while this distinction is helpful from the perspective of computer maintenance, it is also a source of inefficiency that is not shared by the human brain. Simply speaking, the hardware and software are tightly integrated in the brain-inspired model of computing. In recent years, progress has been made in the area of *neuromorphic computing* [117]. This notion is based on a new chip architecture containing spiking neurons, low-precision synapses, and a scalable communication network. Readers are referred to [117].

12.8 Summary

In this chapter, several advanced topics in deep learning have been discussed. The chapter starts with a discussion of attention mechanisms, which help make a neural network more expressive and generalizable. These mechanisms have been used for image, text, and graph data. Attention mechanisms can also be used to augment computers with external memory in the form of neural Turing machines.

Generative adversarial networks are recent techniques that use an adversarial interaction process between a generative network and a discriminative network in order to generate synthetic samples that are similar to a database of real samples. In addition, by imposing a conditional on the generative process, it is possible to create samples with different types of contexts. These ideas have been used in various types of applications such as text-to-image and image-to-image translation.

12.9 Bibliographic Notes and Software Resources

Early techniques for using attention in neural network training were proposed in [59, 272]. The recurrent models of visual attention in this chapter are based on [349]. The recognition of multiple objects in an image with visual attention is discussed in [16] and the spatial transformer is introduced in [229]. The two most well known models are neural machine translation with attention are discussed in [19, 311]. The ideas of attention have also been extended to image captioning [562]. The use of attention models for text summarization is discussed in [429]. The notion of attention is also useful for focusing on specific parts of the image to enable visual question-answering [411, 561, 564]. Transformer networks that process text without RNNs were proposed in [521] and those that process images without CNNs are discussed in [106]. Transformers have been used for unsupervised natural

language generation models and meta-learning in the form of Google BERT [101] and Open AI's GPT-n [50]. The T5 model was proposed in [400]. Other notable language models include T5 [400], Microsoft Turing Natural Language Generator (NLG) [672], XLNet [565], and the Switch Transformer [120]. The discussion on graph attention networks is adapted from [525].

Neural Turing machines [165] and memory networks [489, 547] are similar architectures that were proposed around the same time. The work in [409] proposes a neural program interpreter, which is a neural network that learns to represent and execute programs. The use of reinforcement learning in neural Turing machines is discussed in [575, 576]. The differentiable neural computer [166] is an enhancement of the neural Turing machine with advanced management of memory allocation and sequential writes.

A survey and a tutorial of adversarial learning methods is given in [62, 679]. Generative adversarial networks (GANs) were proposed in [156], and an excellent tutorial may be found in [152]. Improved training algorithms are discussed in [437]. The main challenges in training adversarial networks have to do with *instability* and *saturation* [12, 13]. Energy-based GANs are proposed in [586], and it is claimed that they have better stability. Adversarial ideas have also been generalized to autoencoder architectures [323]. Generative adversarial networks for images use deconvolutional networks to generate realistic images [97, 398], and therefore the resulting GAN is referred to as a DCGAN. The idea of conditional generative networks and their use in generating objects with context is discussed in [343, 408]. The approach has also been used recently for image to image translation [226, 383, 538]. The use of CGANs for predicting the next frame in a video is discussed in [330]. Generative adversarial networks have also been extended to sequences [571].

The earliest works on competitive learning may be found in [426, 427]. Gersho and Gray [142] provide an excellent overview of vector quantization methods. Vector quantization methods are alternatives to sparse coding techniques [78]. The Kohonen self-organizing feature map is discussed in detail in [258]. Many variants of this basic architecture, such as *neural gas*, are used for incremental learning [131, 328].

A discussion of learning-to-learn methods may be found in [514]. The methods used in this area range widely including Bayesian models [121], neural Turing machines [433, 527], and evolutionary algorithms [566]. Energy-efficient methods for deep learning include the use of binary weights [86, 405], specially designed chips [117], enhanced data reuse [70], and compression mechanisms [178, 179, 224].

Software Resources

The recurrent model for visual attention is available at [649]. The MATLAB code for the attention mechanism for neural machine translation discussed in this chapter (from the original authors) may be found in [650]. Implementations of transformer networks may be found from HuggingFace and Google Brain Trax among others [673, 680, 681]. The implementations of numerous language models are available, such as those of ELMo [678], BERT [674], T5 [675], XLNet [677], and Switch Transformer [676]. Implementations of the Neural Turing Machine in *TensorFlow* may be found in [651, 652]. The two implementations are related because the approach in [652] adopts some of the portions of [651]. An LSTM controller is used in the original implementation. Implementations in *Keras*, *Lasagne*, and *Torch* may be found in [653–655]. Several implementations from *Facebook* on memory networks are available at [656]. An implementation of memory networks in *TensorFlow* nay be found in [657]. An implementation of dynamic memory networks in *Theano* and *Lasagne* is available at [658].

An implementation of DCGAN in *TensorFlow* may be found in [659]. In fact, several variants of the GAN (and other topics discussed in this chapter) are available from this contributor [660]. A *Keras* implementation of the GAN may be found in [661]. Implementations of various types of GANs, including the Wasserstein GAN and the variational autoencoder may be found in [662]. These implementations are executed in *PyTorch* and *TensorFlow*. An implementation of the text-to-image GAN in *TensorFlow* is provided in [663], and this implementation is built on top of the aforementioned DCGAN implementation [659].

12.10 Exercises

1. What are the main differences in the approaches used for training hard-attention and soft-attention models?

2. What is the computational complexity of a self-attention mechanism on a sentence of length n?

3. Discuss how the k-means algorithm is related to competitive learning.

4. Implement a Kohonen self-organizing map with (i) a rectangular lattice, and (ii) a hexagonal lattice.

5. Consider a two-player game like GANs with objective function $f(x, y)$, and we want to compute $\min_x \max_y f(x, y)$. Discuss the relationship between $\min_x \max_y f(x, y)$ and $\max_y \min_x f(x, y)$. When are they equal?

6. Consider the function $f(x, y) = \sin(x + y)$, where we are trying to minimize $f(x, y)$ with respect to x and maximize with respect to y. Implement the alternating process of gradient descent and ascent discussed in the book for GANs to optimize this function. Do you always get the same solution over different starting points?

7. Discuss how the Luong attention model can be used for self attention in language modeling. Provide an example of when attention might help predictions.

8. Discuss how you can force the localization network and grid generation in the spatial transformer to ensure that (a) only rotations are allowed without translation, reflection, or scaling, and (b) scalings of equal magnitude in all directions are allowed along with rotations and translations, but reflections are not allowed.

9. Construct a coordinate transformation matrix for the spatial transformer that causes the lower half of the transformed image to be blank.

10. Consider an RBF classifier that uses sampled data points as the means of the RBF layer. Discuss why it can be considered a simple example of an attention mechanism. Relate attention methods to kernel classifiers in general.

11. Discuss the behavior of the softmax function on a vector whose components have large magnitudes. What are the goals of scaling in scaled dot-product attention? Can you think of ways of optimizing the scaling for a specific training data set?

12. Propose an architecture for combining the channel-wise attention of SENet and the spatial attention of the spatial transformer.

13. What changes would you make to the transformer architecture to make it work for sets instead of sequences?

Bibliography

[1] D. Ackley, G. Hinton, and T. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1), pp. 147–169, 1985.

[2] C. Aggarwal. Data classification: Algorithms and applications, *CRC Press*, 2014.

[3] C. Aggarwal. Data mining: The textbook. *Springer*, 2015.

[4] C. Aggarwal. Recommender systems: The textbook. *Springer*, 2016.

[5] C. Aggarwal. Outlier analysis. *Springer*, 2017.

[6] C. Aggarwal. Linear algebra and optimization for machine learning: A Textbook. *Springer*, 2020.

[7] C. Aggarwal. Machine learning for text. *Springer*, 2018.

[8] R. Ahuja, T. Magnanti, and J. Orlin. Network flows: Theory, algorithms, and applications. *Prentice Hall*, 1993.

[9] E. Aljalbout, V. Golkov, Y. Siddiqui, and D. Cremers. Clustering with deep learning: Taxonomy and new methods. *arXiv:1801.07648*, 2018. https://arxiv.org/abs/1801.07648

[10] R. Al-Rfou, B. Perozzi, and S. Skiena. Polyglot: Distributed word representations for multilingual nlp. *arXiv:1307.1662*, 2013. https://arxiv.org/abs/1307.1662

[11] D. Amodei *at al.* Concrete problems in AI safety. *arXiv:1606.06565*, 2016. https://arxiv.org/abs/1606.06565

[12] M. Arjovsky and L. Bottou. Towards principled methods for training generative adversarial networks. *arXiv:1701.04862*, 2017. https://arxiv.org/abs/1701.04862

[13] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *arXiv:1701.07875*, 2017. https://arxiv.org/abs/1701.07875

[14] J. Ba and R. Caruana. Do deep nets really need to be deep? *NeurIPS*, pp. 2654–2662, 2014.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0

[15] J. Ba, J. Kiros, and G. Hinton. Layer normalization. *arXiv:1607.06450*, 2016. https://arxiv.org/abs/1607.06450

[16] J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv: 1412.7755*, 2014. https://arxiv.org/abs/1412.7755

[17] A. Babenko, A. Slesarev, A. Chigorin, and V. Lempitsky. Neural codes for image retrieval. *arXiv:1404.1777*, 2014. https://arxiv.org/abs/1404.1777

[18] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt. Sequential deep learning for human action recognition. *International Workshop on Human Behavior Understanding*, pp. 29–39, 2011.

[19] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015. https://arxiv.org/abs/1409.0473

[20] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv:1611.02167*, 2016. https://arxiv.org/abs/1611.02167

[21] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri. Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11), pp. 937–946, 1999.

[22] N. Ballas, L. Yao, C. Pal, and A. Courville. Delving deeper into convolutional networks for learning video representations. *arXiv:1511.06432*, 2015. https://arxiv.org/abs/1511.06432

[23] J. Baxter, A. Tridgell, and L. Weaver. Knightcap: a chess program that learns by combining td (lambda) with game-tree search. *arXiv cs/9901002*, 1999. https://arxiv.org/abs/cs/9901002

[24] M. Bazaraa, H. Sherali, and C. Shetty. Nonlinear programming: theory and algorithms. *John Wiley and Sons*, 2013.

[25] S. Becker, and Y. LeCun. Improving the convergence of back-propagation learning with second order methods. *Proceedings of the 1988 connectionist models summer school*, pp. 29–37, 1988.

[26] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, pp. 253–279, 2013.

[27] R. E. Bellman. Dynamic Programming. *Princeton University Press*, 1957.

[28] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1), pp. 1–127, 2009.

[29] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE TPAMI*, 35(8), pp. 1798–1828, 2013.

[30] Y. Bengio and O. Delalleau. Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6), pp. 1601–1621, 2009.

[31] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *NeurIPS*, 19, 153, 2007.

[32] Y. Bengio, N. Le Roux, P. Vincent, O. Delalleau, and P. Marcotte. Convex neural networks. *NeurIPS*, pp. 123–130, 2005.

[33] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. *ICML*, 2009.

[34] Y. Bengio, L. Yao, G. Alain, and P. Vincent. Generalized denoising auto-encoders as generative models. *NeurIPS*, pp. 899–907, 2013.

[35] J. Bergstra *et al.* Theano: A CPU and GPU math compiler in Python. *Python in Science Conference*, 2010.

[36] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyper-parameter optimization. *NeurIPS*, pp. 2546–2554, 2011.

[37] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13, pp. 281–305, 2012.

[38] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML*, pp. 115–123, 2013.

[39] D. Bertsekas. Nonlinear programming *Athena Scientific*, 1999.

[40] C. M. Bishop. Pattern recognition and machine learning. *Springer*, 2007.

[41] C. M. Bishop. Neural networks for pattern recognition. *Oxford University Press*, 1995.

[42] C. M Bishop. Improving the generalization properties of radial basis function neural networks. *Neural Computation*, 3(4), pp. 579–588, 1991.

[43] C. M. Bishop. Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1),pp. 108–116, 1995.

[44] C. M. Bishop, M. Svensen, and C. K. Williams. GTM: A principled alternative to the self-organizing map. *NeurIPS*, pp. 354–360, 1997.

[45] M. Bojarski *et al.* End to end learning for self-driving cars. *arXiv:1604.07316*, 2016. https://arxiv.org/abs/1604.07316

[46] M. Bojarski *et al.* Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. *arXiv:1704.07911*, 2017. https://arxiv.org/abs/1704.07911

[47] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4), pp. 291–294, 1988.

[48] L. Breiman. Random forests. *Machine Learning*, 45(1), pp. 5–32, 2001.

[49] L. Breiman. Bagging predictors. *Machine Learning*, 24(2), pp. 123–140, 1996.

[50] T. Brown *et al.* Language models are few-shot learners. *arXiv:2005.14165*, 2020. https://arxiv.org/abs/2005.14165

[51] D. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2, pp. 321–355, 1988.

[52] C. Browne *et al.* A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), pp. 1–43, 2012.

[53] T. Brox and J. Malik. Large displacement optical flow: descriptor matching in variational motion estimation. *IEEE TPAMI*, 33(3), pp. 500–513, 2011.

[54] A. Bryson. A gradient method for optimizing multi-stage allocation processes. *Harvard University Symposium on Digital Computers and their Applications*, 1961.

[55] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. *KDD*, pp. 535–541, 2006.

[56] P. Bühlmann and B. Yu. Analyzing bagging. *Annals of Statistics*, pp. 927–961, 2002.

[57] M. Buhmann. Radial Basis Functions: Theory and implementations. *Cambridge University Press*, 2003.

[58] Y. Burda, R. Grosse, and R. Salakhutdinov. Importance weighted autoencoders. *arXiv:1509.00519*, 2015. https://arxiv.org/abs/1509.00519

[59] N. Butko and J. Movellan. I-POMDP: An infomax model of eye movement. *IEEE International Conference on Development and Learning*, pp. 139–144, 2008.

[60] Y. Cao, Y. Chen, and D. Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1), 54–66, 2015.

[61] M. Carreira-Perpinan and G. Hinton. On Contrastive Divergence Learning. *AISTATS*, 10, pp. 33–40, 2005.

[62] A. Chakraborty *et al.* Adversarial attacks and defences: A survey. *arXiv:1810.00069* 2018. https://arxiv.org/abs/1810.00069

[63] S. Chang, W. Han, J. Tang, G. Qi, C. Aggarwal, and T. Huang. Heterogeneous network embedding via deep architectures. *KDD*, pp. 119–128, 2015.

[64] J. Chen, S. Sathe, C. Aggarwal, and D. Turaga. Outlier detection with autoencoder ensembles. *SDM Conference*, 2017.

[65] J. Chen, J. Zhu, and L. Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv:1710.10568*, 2017. https://arxiv.org/abs/1710.10568

[66] J. Chen, T. Ma, and C. Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv:1801.10247*, 2018. https://arxiv.org/abs/1801.10247

[67] S. Chen, C. Cowan, and P. Grant. Orthogonal least-squares learning algorithm for radial basis function networks. *IEEE TNNLS*, 2(2), pp. 302–309, 1991.

[68] W. Chen *et al.* Compressing neural networks with the hashing trick. *ICML*, 2015.

[69] Y. Chen and M. Zaki. KATE: K-Competitive Autoencoder for Text. *KDD*, 2017.

[70] Y. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1), pp. 127–138, 2017.

[71] R. Child *et al.* Generating long sequences with sparse transformers. *arXiv:1904.10509*, 2019.

[72] K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP*, 2014. https://arxiv.org/abs/1406.1078

[73] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio. Attention-based models for speech recognition. *NeurIPS*, pp. 577–585, 2015.

[74] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv:1412.3555*, 2014. https://arxiv.org/abs/1412.3555

[75] D. Ciresan, U. Meier, L. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12), pp. 3207–3220, 2010.

[76] C. Clark and A. Storkey. Training deep convolutional neural networks to play go. *ICML*, pp. 1766–1774, 2015.

[77] A. Coates, B. Huval, T. Wang, D. Wu, A. Ng, and B. Catanzaro. Deep learning with COTS HPC systems. *ICML*, pp. 1337–1345, 2013.

[78] A. Coates and A. Ng. The importance of encoding versus training with sparse coding and vector quantization. *ICML*, pp. 921–928, 2011.

[79] A. Coates and A. Ng. Learning feature representations with k-means. *Neural networks: Tricks of the Trade*, Springer, pp. 561–580, 2012.

[80] A. Coates, A. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. *AAAI*, pp. 215–223, 2011.

[81] R. Collobert *et al.* Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12, pp. 2493–2537, 2011.

[82] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. *ICML*, pp. 160–167, 2008.

[83] J. Connor, R. Martin, and L. Atlas. Recurrent neural networks and robust time series prediction. *IEEE TNNLS*, 5(2), pp. 240–254, 1994.

[84] T. Cooijmans, N. Ballas, C. Laurent, C. Gulcehre, and A. Courville. Recurrent batch normalization. *arXiv:1603.09025*, 2016. https://arxiv.org/abs/1603.09025

[85] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3), pp. 273–297, 1995.

[86] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. *arXiv:1511.00363*, 2015. https://arxiv.org/abs/1511.00363

[87] T. Cover. Geometrical and statistical properties of systems of linear inequalities with applications to pattern recognition. *IEEE Transactions on Electronic Computers*, pp. 326–334, 1965.

[88] D. Cox and N. Pinto. Beyond simple features: A large-scale feature search approach to unconstrained face recognition. *Face and Gesture Recognition*, pp. 8–15, 2011.

[89] G. Dahl, R. Adams, and H. Larochelle. Training restricted Boltzmann machines on word observations. *arXiv:1202.5695*, 2012. https://arxiv.org/abs/1202.5695

[90] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *CVPR*, pp. 886–893, 2005.

[91] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *NeurIPS*, pp. 2933–2941, 2014.

[92] N. de Freitas. Machine Learning, University of Oxford (Course Video), 2013. https://www.youtube.com/watch?v=w2OtwL5T1ow&list=PLE6Wd9FR--EdyJ5lbFl8UuGjecvVw66F6

[93] N. de Freitas. Deep Learning, University of Oxford (Course Video), 2015. https://www.youtube.com/watch?v=PlhFWT7vAEw&list=PLjK8ddCbDMphIMSXn-w1IjyYpHU3DaUYw

[94] J. Dean *et al.* Large scale distributed deep networks. *NeurIPS*, 2012.

[95] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *NeurIPS*, pp. 3844–3852, 2016.

[96] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas. Predicting parameters in deep learning. *NeurIPS*, pp. 2148–2156, 2013.

[97] E. Denton, S. Chintala, and R. Fergus. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. *NeurIPS*, pp. 1466–1494, 2015.

[98] G. Desjardins, K. Simonyan, and R. Pascanu. Natural neural networks. *NeurIPS*, pp. 2071–2079, 2015.

[99] F. Despagne and D. Massart. Neural networks in multivariate calibration. *Analyst*, 123(11), pp. 157R–178R, 1998.

[100] T. Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv:1511.04561*, 2015. https://arxiv.org/abs/1511.04561

[101] J. Devlin *et al.* Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018. https://arxiv.org/abs/1810.04805

[102] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. *CVPR*, pp. 2625–2634, 2015.

[103] G. Dorffner. Neural networks for time series processing. *Neural Network World*, 1996.

[104] C. Dos Santos and M. Gatti. Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. *COLING*, pp. 69–78, 2014.

[105] A. Dosovitskiy and T. Brox. Inverting visual representations with convolutional networks. *CVPR Conference*, pp. 4829–4837, 2016.

[106] A. Dosovitsky *et al.* An image is worth 16×16 words: Transformers for image recognition at scale. *arXiv:2010.11929*, 2020. https://arxiv.org/abs/2010.11929

[107] K. Doya. Bifurcations of recurrent neural networks in gradient descent learning. *IEEE TNNLS*, 1, pp. 75–80, 1993.

[108] C. Doersch. Tutorial on variational autoencoders. *arXiv:1606.05908*, 2016. https://arxiv.org/abs/1606.05908

[109] H. Drucker and Y. LeCun. Improving generalization performance using double backpropagation. *IEEE TNNLS*, 3(6), pp. 991–997, 1992.

[110] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, pp. 2121–2159, 2011.

[111] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016. https://arxiv.org/abs/1603.07285

[112] A. Elkahky, Y. Song, and X. He. A multi-view deep learning approach for cross domain user modeling in recommendation systems. *WWW Conference*, pp. 278–288, 2015.

[113] J. Elman. Finding structure in time. *Cognitive Science*, 14(2), pp. 179–211, 1990.

[114] J. Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48, pp. 781–799, 1993.

[115] T. Elsken, J. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55), pp. 1–21, 2019.

[116] D. Erhan *et al.* Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11, pp. 625–660, 2010.

[117] S. Essar *et al.* Convolutional neural networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Science of the USA*, 113(41), pp. 11441–11446, 2016.

[118] A. Fader, L. Zettlemoyer, and O. Etzioni. Paraphrase-Driven Learning for Open Question Answering. *ACL*, pp. 1608–1618, 2013.

[119] W. Fan *et al.* Graph neural networks for social recommendation. *WWW*, 2019.

[120] W. Fedus, B. Zoph, and N. Shazeer. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv:2101.03961*, 2021. https://arxiv.org/abs/2101.03961

[121] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE TPAMI*, 28(4), pp. 594–611, 2006.

[122] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE TPAMI*, 32(9), pp. 1627–1645, 2010.

[123] A. Fader, L. Zettlemoyer, and O. Etzioni. Open question answering over curated and extracted knowledge bases. *KDD*, 2014.

[124] A. Fischer and C. Igel. An introduction to restricted Boltzmann machines. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pp. 14–36, 2012.

[125] R. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7: pp. 179–188, 1936.

[126] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv:1803.03635*, 2018. https://arxiv.org/abs/1803.03635

[127] Y. Freund and R. Schapire. A decision-theoretic generalization of online learning and application to boosting. *Computational Learning Theory*, pp. 23–37, 1995.

[128] Y. Freund and R. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3), pp. 277–296, 1999.

[129] Y. Freund and D. Haussler. Unsupervised learning of distributions on binary vectors using two layer networks. *Technical report*, Santa Cruz, CA, USA, 1994

[130] B. Fritzke. Fast learning with incremental RBF networks. *Neural Processing Letters*, 1(1), pp. 2–5, 1994.

[131] B. Fritzke. A growing neural gas network learns topologies. *NeurIPS*, pp. 625–632, 1995.

[132] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4), pp. 193–202, 1980.

[133] S. Gallant. Perceptron-based learning algorithms. *IEEE TNNLS*, 1(2), pp. 179–191, 1990.

[134] S. Gallant. Neural network learning and expert systems. *MIT Press*, 1993.

[135] H. Gao, H. Yuan, Z. Wang, and S. Ji. Pixel Deconvolutional Networks. *arXiv:1705.06820*, 2017. https://arxiv.org/abs/1705.06820

[136] H. Gao and S. Ji. Graph u-nets. *arXiv:1905.05178*, 2019. http://proceedings.mlr.press/v97/gao19a/gao19a.pdf
Code: https://github.com/divelab/gunet

[137] L. Gatys, A. S. Ecker, and M. Bethge. Texture synthesis using convolutional neural networks. *NeurIPS*, pp. 262–270, 2015.

[138] L. Gatys, A. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. *CVPR*, pp. 2414–2423, 2015.

[139] H. Gavin. The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems, 2011. http://people.duke.edu/~hpgavin/ce281/lm.pdf

[140] P. Gehler, A. Holub, and M. Welling. The Rate Adapting Poisson (RAP) model for information retrieval and object recognition. *ICML*, 2006.

[141] S. Gelly *et al.* The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55, pp. 106–113, 2012.

[142] A. Gersho and R. M. Gray. Vector quantization and signal compression. *Springer Science and Business Media*, 2012.

[143] A. Ghodsi. STAT 946: Topics in Probability and Statistics: Deep Learning, *University of Waterloo*, Fall 2015. https://www.youtube.com/watch?v=fyAZszlPphs&list=PLehuLRPyt1Hyi78UOkMPWCGRxGcA9NVOE

[144] W. Gilks, S. Richardson, and D. Spiegelhalter. Markov chain Monte Carlo in practice.*CRC Press*, 1995.

[145] J. Gilmer *et al.* Neural message passing for quantum chemistry. *ICML*, 2017.

[146] F. Girosi and T. Poggio. Networks and the best approximation property. *Biological Cybernetics*, 63(3), pp. 169–176, 1990.

[147] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, pp. 249–256, 2010.

[148] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. *AISTATS*, 15(106), 2011.

[149] P. Glynn. Likelihood ratio gradient estimation: an overview, *Proceedings of the 1987 Winter Simulation Conference*, pp. 366–375, 1987.

[150] Y. Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research (JAIR)*, 57, pp. 345–420, 2016.

[151] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv:1412.6572*, 2014. https://arxiv.org/abs/1412.6572

[152] I. Goodfellow. NIPS 2016 tutorial: Generative adversarial networks. *arXiv:1701.00160*, 2016. https://arxiv.org/abs/1701.00160

[153] I. Goodfellow, O. Vinyals, and A. Saxe. Qualitatively characterizing neural network optimization problems. *ICLR*, 2015. https://arxiv.org/abs/1412.6544

[154] I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. *MIT Press*, 2016.

[155] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv:1302.4389*, 2013.

[156] I. Goodfellow *et al.* Generative adversarial nets. *NeurIPS*, 2014.

[157] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. *Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6645–6649, 2013.

[158] A. Graves. Generating sequences with recurrent neural networks. *arXiv:1308.0850*, 2013. https://arxiv.org/abs/1308.0850

[159] A. Graves. Supervised sequence labelling with recurrent neural networks *Springer*, 2012.

[160] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. *ICML*, pp. 369–376, 2006.

[161] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE TPAMI*, 31(5), pp. 855–868, 2009.

[162] A. Graves and J. Schmidhuber. Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures. *Neural Networks*, 18(5-6), pp. 602–610, 2005.

[163] A. Graves and J. Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. *NeurIPS*, pp. 545–552, 2009.

[164] A. Graves and N. Jaitly. Towards End-To-End Speech Recognition with Recurrent Neural Networks. *ICML*, pp. 1764–1772, 2014.

[165] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv:1410.5401*, 2014. https://arxiv.org/abs/1410.5401

[166] A. Graves *et al.* Hybrid computing using a neural network with dynamic external memory. *Nature*, 538.7626, pp. 471–476, 2016.

[167] K. Greff, R. K. Srivastava, J. Koutnik, B. Steunebrink, and J. Schmidhuber. LSTM: A search space odyssey. *IEEE TNNLS*, 2016.

[168] K. Greff, R. K. Srivastava, and J. Schmidhuber. Highway and residual networks learn unrolled iterative estimation. *arXiv:1612.07771*, 2016. https://arxiv.org/abs/1612.07771

[169] I. Grondman, L. Busoniu, G. A. Lopes, and R. Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics*, 42(6), pp. 1291–1307, 2012.

[170] R. Girshick, F. Iandola, T. Darrell, and J. Malik. Deformable part models are convolutional neural networks. *CVPR*, pp. 437–446, 2015.

[171] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. *KDD*, pp. 855–864, 2016.

[172] X. Guo, S. Singh, H. Lee, R. Lewis, and X. Wang. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. *NeurIPS*, pp. 3338–3346, 2014.

[173] M. Gutmann and A. Hyvarinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *AISTATS*, 1(2), pp. 6, 2010.

[174] R. Hahnloser and H. S. Seung. Permitted and forbidden sets in symmetric threshold-linear networks. *NeurIPS*, pp. 217–223, 2001.

[175] W. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: methods and applications. *arXiv:1709.05584*, 2017. https://arxiv.org/abs/1709.05584

[176] W. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *NeurIPS*, pp. 1024–1034, 2017.

[177] W. Hamilton, J. Leskovec, R. Ying, and R. Sosic. Representation learning on networks, *WWW Tutorial*, 2018. http://snap.stanford.edu/proj/embeddings-www/

[178] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally. EIE: Efficient Inference Engine for Compressed Neural Network. *ACM SIGARCH Computer Architecture News*, 44(3), pp. 243–254, 2016.

[179] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural networks. *NeurIPS*, pp. 1135–1143, 2015.

[180] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE TPAMI*, 12(10), pp. 993–1001, 1990.

[181] M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. *ICML*, pp. 1225–1234, 2006.

[182] B. Hariharan, P. Arbelaez, R. Girshick, and J. Malik. Simultaneous detection and segmentation. *arXiv:1407.1808*, 2014. https://arxiv.org/abs/1407.1808

[183] E. Hartman, J. Keeler, and J. Kowalski. Layered neural networks with Gaussian hidden units as universal approximations. *Neural Computation*, 2(2), pp. 210–215, 1990.

[184] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. *AAAI*, 2016.

[185] B. Hassibi and D. Stork. Second order derivatives for network pruning: Optimal brain surgeon. *NeurIPS*, 1993.

[186] D. Hassabis, D. Kumaran, C. Summerfield, and M. Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2), pp. 245–258, 2017.

[187] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning. *Springer*, 2009.

[188] T. Hastie and R. Tibshirani. Generalized additive models. *CRC Press*, 1990.

[189] T. Hastie, R. Tibshirani, and M. Wainwright. Statistical learning with sparsity: the lasso and generalizations. *CRC Press*, 2015.

[190] M. Havaei *et al.* Brain tumor segmentation with deep neural networks. *Medical Image Analysis*, 35, pp. 18–31, 2017.

[191] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier detection using replicator neural networks. *International Conference on Data Warehousing and Knowledge Discovery*, pp. 170–180, 2002.

[192] S. Haykin. Neural networks and learning machines. *Pearson*, 2008.

[193] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *ICCV*, pp. 1026–1034, 2015.

[194] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CVPR*, pp. 770–778, 2016.

[195] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. *ECCV*, pp. 630–645, 2016.

[196] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. S. Chua. Neural collaborative filtering. *WWW*, pp. 173–182, 2017.

[197] N. Heess *et al.* Emergence of Locomotion Behaviours in Rich Environments. *arXiv:1707.02286*, 2017. https://arxiv.org/abs/1707.02286
Video 1 at: https://www.youtube.com/watch?v=hx_bgoTF7bs
Video 2 at: https://www.youtube.com/watch?v=gn4nRCC9TwQ&feature=youtu.be

[198] M. Henaff, J. Bruna, and Y. LeCun. Deep convolutional networks on graph-structured data. *arXiv:1506.05163*, 2015. https://arxiv.org/abs/1506.05163

[199] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952.

[200] G. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1–3), pp. 185–234, 1989.

[201] G. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8), pp. 1771–1800, 2002.

[202] G. Hinton. To recognize shapes, first learn to generate images. *Progress in Brain Research*, 165, pp. 535–547, 2007.

[203] G. Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1), 926, 2010.

[204] G. Hinton. Neural networks for machine learning, *Coursera Video*, 2012.

[205] G. Hinton, P. Dayan, B. Frey, and R. Neal. The wake–sleep algorithm for unsupervised neural networks. *Science*, 268(5214), pp. 1158–1162, 1995.

[206] G. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7), pp. 1527–1554, 2006.

[207] G. Hinton and T. Sejnowski. Learning and relearning in Boltzmann machines. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, 1986.

[208] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313, (5766), pp. 504–507, 2006.

[209] G. Hinton and R. Salakhutdinov. Replicated softmax: an undirected topic model. *NeurIPS*, pp. 1607–1614, 2009.

[210] G. Hinton and R. Salakhutdinov. A better way to pretrain deep Boltzmann machines. *NeurIPS*, pp. 2447–2455, 2012.

[211] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012. https://arxiv.org/abs/1207.0580

[212] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *NeurIPS Workshop*, 2014.

[213] R. Hochberg. Matrix Multiplication with CUDA: A basic introduction to the CUDA programming model. *Unpublished manuscript*, 2012. http://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf

[214] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), pp. 1735–1785, 1997.

[215] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, *A Field Guide to Dynamical Recurrent Neural Networks*, IEEE Press, 2001.

[216] T. Hofmann. Probabilistic latent semantic indexing. *ACM SIGIR Conference*, pp. 50–57, 1999.

[217] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *National Academy of Sciences of the USA*, 79(8), pp. 2554–2558, 1982.

[218] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), pp. 359–366, 1989.

[219] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *CVPR*, pp. 7132–7141, 2018.

[220] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. *IEEE International Conference on Data Mining*, pp. 263–272, 2008.

[221] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Weinberger. Deep networks with stochastic depth. *ECCV*, pp. 646–661, 2016.

[222] G. Huang, Z. Liu, K. Weinberger, and L. van der Maaten. Densely connected convolutional networks. *arXiv:1608.06993*, 2016. https://arxiv.org/abs/1608.06993

[223] D. Hubel and T. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 124(3), pp. 574–591, 1959.

[224] F. Iandola *et al.* SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size. *arXiv:1602.07360*, 2016. https://arxiv.org/abs/1602.07360

[225] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167*, 2015. https://arxiv.org/abs/1502.03167

[226] P. Isola, J. Zhu, T. Zhou, and A. Efros. Image-to-image translation with conditional adversarial networks. *arXiv:1611.07004*, 2016. https://arxiv.org/abs/1611.07004

[227] M. Iyyer, J. Boyd-Graber, L. Claudino, R. Socher, and H. Daume III. A Neural Network for Factoid Question Answering over Paragraphs. *EMNLP*, 2014.

[228] R. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4), pp. 295–307, 1988.

[229] M. Jaderberg *et al.* Spatial transformer networks. *NeurIPS*, pp. 2017–2025, 2015.

[230] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks – with an erratum note. *German National Research Center for Information Technology GMD Technical Report*, 148(34), 13, 2001.

[231] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304, pp. 78–80, 2004.

[232] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? *ICCV*, 2009.

[233] S. Ji, W. Xu, M. Yang, and K. Yu. 3D convolutional neural networks for human action recognition. *IEEE TPAMI*, 35(1), pp. 221–231, 2013.

[234] Y. Jia *et al.* Caffe: Convolutional architecture for fast feature embedding. *ACM International Conference on Multimedia*, 2014.

[235] C. Johnson. Logistic matrix factorization for implicit feedback data. *NeurIPS*, 2014.

[236] J. Johnson, A. Karpathy, and L. Fei-Fei. Densecap: Fully convolutional localization networks for dense captioning. *CVPR*, pp. 4565–4574, 2015.

[237] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. *ECCV*, pp. 694–711, 2015.

[238] R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv:1412.1058*, 2014. https://arxiv.org/abs/1412.1058

[239] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. *ICML*, pp. 2342–2350, 2015.

[240] S. Kakade. A natural policy gradient. *NeurIPS*, pp. 1057–1063, 2002.

[241] N. Kalchbrenner and P. Blunsom. Recurrent continuous translation models. *EMNLP*, 3, 39, pp. 413, 2013.

[242] H. Kandel, J. Schwartz, T. Jessell, S. Siegelbaum, and A. Hudspeth. Principles of neural science. *McGraw Hill*, 2012.

[243] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv:1506.02078*, 2015. https://arxiv.org/abs/1506.02078

[244] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. *CVPR*, pp. 725–1732, 2014.

[245] A. Karpathy. The unreasonable effectiveness of recurrent neural networks, *Blog post*, 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[246] A. Karpathy, J. Johnson, and L. Fei-Fei. Stanford University Class CS321n: Convolutional neural networks for visual recognition, 2016. http://cs231n.github.io/

[247] H. J. Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10), pp. 947–954, 1960.

[248] F. Khan, B. Mutlu, and X. Zhu. How do humans teach: On curriculum learning and teaching dimension. *NeurIPS*, pp. 1449–1457, 2011.

[249] T. Kietzmann, P. McClure, and N. Kriegeskorte. Deep Neural Networks In Computational Neuroscience. *bioRxiv, 133504*, 2017. https://www.biorxiv.org/content/early/2017/05/04/133504

[250] Y. Kim. Convolutional neural networks for sentence classification. *arXiv:1408.5882*, 2014. https://arxiv.org/abs/1408.5882

[251] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014. https://arxiv.org/abs/1412.6980

[252] D. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv:1312.6114*, 2013. https://arxiv.org/abs/1312.6114

[253] T. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907*, 2016. https://arxiv.org/abs/1609.02907
Code: https://github.com/tkipf/gcn

[254] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220, pp. 671–680, 1983.

[255] J. Kivinen and M. Warmuth. The perceptron algorithm vs. winnow: linear vs. logarithmic mistake bounds when few input variables are relevant. *Computational Learning Theory*, pp. 289–296, 1995.

[256] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. *ECML*, pp. 282–293, 2006.

[257] R. Kohavi and D. Wolpert. Bias plus variance decomposition for zero-one loss functions. *ICML*, 1996.

[258] T. Kohonen. Self-organizing maps, *Springer*, 2001.

[259] D. Koller and N. Friedman. Probabilistic graphical models: principles and techniques. *MIT Press*, 2009.

[260] E. Kong and T. Dietterich. Error-correcting output coding corrects bias and variance. *ICML*, pp. 313–321, 1995.

[261] Y. Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM TKDD Journal*, 4(1), 1, 2010.

[262] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997*, 2014. https://arxiv.org/abs/1404.5997

[263] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *NeurIPS*, pp. 1097–1105. 2012.

[264] M. Kubat. Decision trees can initialize radial-basis function networks. *IEEE TNNLS*, 9(5), pp. 813–821, 1998.

[265] M. Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv:1509.01549*, 2015. https://arxiv.org/abs/1509.01549

[266] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent Convolutional Neural Networks for Text Classification. *AAAI*, pp. 2267–2273, 2015.

[267] B. Lake, T. Ullman, J. Tenenbaum, and S. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, pp. 1–101, 2016.

[268] H. Larochelle. Neural Networks (Course). Universite de Sherbrooke, 2013. https://www.youtube.com/watch?v=SGZ6BttHMPw&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH

[269] H. Larochelle and Y. Bengio. Classification using discriminative restricted Boltzmann machines. *ICML*, pp. 536–543, 2008.

[270] H. Larochelle, M. Mandel, R. Pascanu, and Y. Bengio. Learning algorithms for the classification restricted Boltzmann machine. *Journal of Machine Learning Research*, 13, pp. 643–669, 2012.

[271] H. Larochelle and I. Murray. The neural autoregressive distribution estimator. *International Conference on Artificial Intelligence and Statistics*, pp. 29–37, 2011.

[272] H. Larochelle and G. E. Hinton. Learning to combine foveal glimpses with a third-order Boltzmann machine. *NeurIPS*, 2010.

[273] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. *ICML*, pp. 473–480, 2007.

[274] G. Larsson, M. Maire, and G. Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *arXiv:1605.07648*, 2016. https://arxiv.org/abs/1605.07648

[275] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE TNNLS*, 8(1), pp. 98–113, 1997.

[276] Q. Le *et al.* Building high-level features using large scale unsupervised learning. *ICASSP*, 2013.

[277] Q. Le, N. Jaitly, and G. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv:1504.00941*, 2015. https://arxiv.org/abs/1504.00941

[278] Q. Le and T. Mikolov. Distributed representations of sentences and documents. *ICML*, pp. 1188–196, 2014.

[279] Q. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Ng, On optimization methods for deep learning. *ICML*, pp. 265–272, 2011.

[280] Q. Le, W. Zou, S. Yeung, and A. Ng. Learning hierarchical spatio-temporal features for action recognition with independent subspace analysis. *CVPR*, 2011.

[281] Y. LeCun. Modeles connexionnistes de l'apprentissage. *Doctoral Dissertation*, Universite Paris, 1987.

[282] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks*, 3361(10), 1995.

[283] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553), pp. 436–444, 2015.

[284] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. in G. Orr and K. Muller (eds.) *Neural Networks: Tricks of the Trade*, Springer, 1998.

[285] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), pp. 2278–2324, 1998.

[286] Y. LeCun, S. Chopra, R. M. Hadsell, M. A. Ranzato, and F.-J. Huang. A tutorial on energy-based learning. *Predicting Structured Data*, MIT Press, pp. 191–246,, 2006.

[287] Y. LeCun, C. Cortes, and C. Burges. The MNIST database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist/

[288] Y. LeCun, J. Denker, and S. Solla. Optimal brain damage. *NeurIPS*, pp. 598–605, 1990.

[289] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional networks and applications in vision. *IEEE International Symposium on Circuits and Systems*, pp. 253–256, 2010.

[290] H. Lee, C. Ekanadham, and A. Ng. Sparse deep belief net model for visual area V2. *NeurIPS*, 2008.

[291] H. Lee, R. Grosse, B. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *ICML*, pp. 609–616, 2009.

[292] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39), pp. 1–40, 2016.
Video at: https://sites.google.com/site/visuomotorpolicy/

[293] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. *NeurIPS*, pp. 2177–2185, 2014.

[294] O. Levy, Y. Goldberg, and I. Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3, pp. 211–225, 2015.

[295] W. Levy and R. Baxter. Energy efficient neural codes. *Neural Computation*, 8(3), pp. 531–543, 1996.

[296] M. Lewis *et al.* Deal or No Deal? End-to-End Learning for Negotiation Dialogues. *arXiv:1706.05125*, 2017. https://arxiv.org/abs/1706.05125

[297] G. Li, M. Muller, A. Thabet, and B. Ghanem. Deepgcns: Can gcns go as deep as cnns? *CVPR*, pp. 9267-9276, 2019.

[298] J. Li, W. Monroe, A. Ritter, M. Galley,, J. Gao, and D. Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv:1606.01541*, 2016. https://arxiv.org/abs/1606.01541

[299] L. Li, W. Chu, J. Langford, and R. Schapire. A contextual-bandit approach to personalized news article recommendation. *WWW Conference*, pp. 661–670, 2010.

[300] Q. Li, Z. Han, and X.-M. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *AAAI*, 2018.

[301] Y. Li. Deep reinforcement learning: An overview. *arXiv:1701.07274*, 2017. https://arxiv.org/abs/1701.07274

[302] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv:1511.05493*, 2015. https://arxiv.org/abs/1511.05493

[303] Q. Liao, K. Kawaguchi, and T. Poggio. Streaming normalization: Towards simpler and more biologically-plausible normalizations for online and recurrent learning. *arXiv:1610.06160*, 2016. https://arxiv.org/abs/1610.06160

[304] D. Liben-Nowell, and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7), pp. 1019–1031, 2007.

[305] L.-J. Lin. Reinforcement learning for robots using neural networks. *Technical Report*, DTIC Document, 1993.

[306] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv:1312.4400*, 2013. https://arxiv.org/abs/1312.4400

[307] Z. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv:1506.00019*, 2015. https://arxiv.org/abs/1506.00019

[308] J. Lu, J. Yang, D. Batra, and D. Parikh. Hierarchical question-image co-attention for visual question answering. *NeurIPS*, pp. 289–297, 2016.

[309] D. Luenberger and Y. Ye. Linear and nonlinear programming, *Addison-Wesley*, 1984.

[310] M. Lukosevicius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3), pp. 127–149, 2009.

[311] M. Luong, H. Pham, and C. Manning. Effective approaches to attention-based neural machine translation. *arXiv:1508.04025*, 2015. https://arxiv.org/abs/1508.04025

[312] J. Ma, R. P. Sheridan, A. Liaw, G. E. Dahl, and V. Svetnik. Deep neural nets as a method for quantitative structure-activity relationships. *Journal of Chemical Information and Modeling*, 55(2), pp. 263–274, 2015.

[313] Y. Ma, S. Wang, C. Aggarwal, and J. Tang. Graph convolutional networks with eigenpooling. *KDD*, 2019. Code: https://github.com/alge24/eigenpooling

[314] Y. Ma, S. Wang, C. Aggarwal, D. Yin, and J. Tang. Multi-dimensional graph convolutional networks. *SDM Conference*, 2019.

[315] W. Maass, T. Natschlager, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11), pp. 2351–2560, 2002.

[316] L. Maaten and G. E. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9, pp. 2579–2605, 2008.

[317] D. J. MacKay. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3), pp. 448–472, 1992.

[318] C. Maddison, A. Huang, I. Sutskever, and D. Silver. Move evaluation in Go using deep convolutional neural networks. *ICLR*, 2015.

[319] A. Madry *et al.* Towards deep learning models resistant to adversarial attacks. *arXiv:1706.06083*, 2017. https://arxiv.org/abs/1706.06083

[320] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. *CVPR*, pp. 5188–5196, 2015.

[321] A. Makhzani and B. Frey. K-sparse autoencoders. *arXiv:1312.5663*, 2013. https://arxiv.org/abs/1312.5663

[322] A. Makhzani and B. Frey. Winner-take-all autoencoders. *NeurIPS*, pp. 2791–2799, 2015.

[323] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey. Adversarial autoencoders. *arXiv:1511.05644*, 2015. https://arxiv.org/abs/1511.05644

[324] J. Martens. Deep learning via Hessian-free optimization. *ICML*, pp. 735–742, 2010.

[325] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. *ICML*, pp. 1033–1040, 2011.

[326] J. Martens, I. Sutskever, and K. Swersky. Estimating the hessian by back-propagating curvature. *arXiv:1206.6464*, 2016. https://arxiv.org/abs/1206.6464

[327] J. Martens and R. Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. *ICML*, 2015.

[328] T. Martinetz, S. Berkovich, and K. Schulten. 'Neural-gas' network for vector quantization and its application to time-series prediction. *IEEE TNNLS*, 4(4), pp. 558–569, 1993.

[329] J. Masci, U. Meier, D. Ciresan, and J. Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning*, pp. 52–59, 2011.

[330] M. Mathieu, C. Couprie, and Y. LeCun. Deep multi-scale video prediction beyond mean square error. *arXiv:1511.054*, 2015. https://arxiv.org/abs/1511.05440

[331] P. McCullagh and J. Nelder. Generalized linear models *CRC Press*, 1989.

[332] W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), pp. 115–133, 1943.

[333] G. McLachlan. Discriminant analysis and statistical pattern recognition *John Wiley & Sons*, 2004.

[334] C. Micchelli. Interpolation of scattered data: distance matrices and conditionally positive definite functions. *Constructive Approximations*, 2, pp. 11–22, 1986.

[335] P. Michel, O. Levy, and G. Neubig. Are sixteen heads really better than one? *arXiv:1905.10650*, 2019. https://arxiv.org/abs/1905.10650

[336] T. Mikolov. Statistical language models based on neural networks. *Ph.D. thesis, Brno University of Technology*, 2012.

[337] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013. https://arxiv.org/abs/1301.3781

[338] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato. Learning longer memory in recurrent neural networks. *arXiv:1412.7753*, 2014. https://arxiv.org/abs/1412.7753

[339] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *NeurIPS*, pp. 3111–3119, 2013.

[340] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur. Recurrent neural network based language model. *Interspeech*, Vol 2, 2010.

[341] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4), pp. 235–312, 1990. https://wordnet.princeton.edu/

[342] M. Minsky and S. Papert. Perceptrons. An Introduction to Computational Geometry, *MIT Press*, 1969.

[343] M. Mirza and S. Osindero. Conditional generative adversarial nets. *arXiv:1411.1784*, 2014. https://arxiv.org/abs/1411.1784

[344] A. Mnih and K. Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. *NeurIPS*, pp. 2265–2273, 2013.

[345] A. Mnih and Y. Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv:1206.6426*, 2012. https://arxiv.org/abs/1206.6426

[346] V. Mnih *et al.* Human-level control through deep reinforcement learning. *Nature*, 518 (7540), pp. 529–533, 2015.

[347] V. Mnih *et al.* Playing atari with deep reinforcement learning. *arXiv:1312.5602.*, 2013. https://arxiv.org/abs/1312.5602

[348] V. Mnih *et al.* Asynchronous methods for deep reinforcement learning. *ICML*, pp. 1928–1937, 2016.

[349] V. Mnih, N. Heess, and A. Graves. Recurrent models of visual attention. *NeurIPS*, pp. 2204–2212, 2014.

[350] H. Mobahi and J. Fisher. A theoretical analysis of optimization by Gaussian continuation. *AAAI*, 2015.

[351] G. Montufar. Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, 26(7), pp. 1386–1407, 2014.

[352] G. Montufar and N. Ay. Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. *Neural Computation*, 23(5), pp. 1306–1319, 2011.

[353] J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2), pp. 281–294, 1989.

[354] A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1), pp. 103–130, 1993.

[355] F. Morin and Y. Bengio. Hierarchical Probabilistic Neural Network Language Model. *AISTATS*, pp. 246–252, 2005.

[356] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in Bioinformatics*, pp. 1–11, 2017.

[357] M. Müller, M. Enzenberger, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2, pp. 259–270, 2010.

[358] M. Musavi, W. Ahmed, K. Chan, K. Faris, and D. Hummels. On the training of radial basis function classifiers. *Neural Networks*, 5(4), pp. 595–603, 1992.

[359] V. Nair and G. Hinton. Rectified linear units improve restricted Boltzmann machines. *ICML*, pp. 807–814, 2010.

[360] K. S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE TNNLS*, 1(1), pp. 4–27, 1990.

[361] R. Neal. Connectionist learning of belief networks. *Artificial intelligence*, 1992.

[362] R. Neal. Probabilistic inference using Markov chain Monte Carlo methods. *Technical Report CRG-TR-93-1*, 1993.

[363] R. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2), pp. 125–139, 2001.

[364] Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27, pp. 372–376, 1983.

[365] A. Ng. Sparse autoencoder. *CS294A Lecture notes*, 2011.
https://nlp.stanford.edu/~socherr/sparseAutoencoder_2011new.pdf
https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf

[366] A. Ng and M. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. *Uncertainity in Artificial Intelligence*, pp. 406–415, 2000.

[367] J. Y.-H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. *CVPR*, pp. 4694–4702, 2015.

[368] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Ng. Multimodal deep learning. *ICML*, pp. 689–696, 2011.

[369] A. Nguyen *et al.* Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. *NeurIPS*, pp. 3387–3395, 2016.

[370] J. Nocedal and S. Wright. Numerical optimization. *Springer*, 2006.

[371] S. Nowlan and G. Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4), pp. 473–493, 1992.

[372] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Learning and transferring mid-level image representations using convolutional neural networks. *CVPR*, pp. 1717–1724, 2014.

[373] G. Orr and K.-R. Müller (editors). Neural Networks: Tricks of the Trade, *Springer*, 1998.

[374] M. J. L. Orr. Introduction to radial basis function networks, *University of Edinburgh Technical Report, Centre of Cognitive Science*, 1996. ftp://ftp.cogsci.ed.ac.uk/pub/mjo/intro.ps.Z

[375] L. Ouyang *et al.* Training language models to follow instructions with human feedback. *arXiv*:2203.02155, 2022. https://arxiv.org/abs/2203.02155

[376] N. Papernot *et al.* Practical black-box attacks against machine learning. *ACM CCS Conference*, 2017. https://dl.acm.org/doi/pdf/10.1145/3052973.3053009

[377] N. Papernot *et al.* Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks. *arXiv:1511.04508*, 2015. https://arxiv.org/abs/1511.04508

[378] J. Park and I. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(1), pp. 246–257, 1991.

[379] J. Park and I. Sandberg. Approximation and radial-basis-function networks. *Neural Computation*, 5(2), pp. 305–316, 1993.

[380] O. Parkhi, A. Vedaldi, and A. Zisserman. Deep Face Recognition. *BMVC*, 1(3), pp. 6, 2015.

[381] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. *ICML*, 28, pp. 1310–1318, 2013.

[382] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2012.

[383] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros. Context encoders: Feature learning by inpainting. *CVPR Conference*, 2016.

[384] J. Pennington, R. Socher, and C. Manning. Glove: Global Vectors for Word Representation. *EMNLP*, pp. 1532–1543, 2014.

[385] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. *KDD*, pp. 701-710, 2014.

[386] M. Peters *et al.* Deep contextualized word representations. *arXiv:1802.05365*, 2018. https://arxiv.org/abs/1802.05365

[387] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4), pp. 682–697, 2008.

[388] C. Peterson and J. Anderson. A mean field theory learning algorithm for neural networks. *Complex Systems*, 1(5), pp. 995-1019, 1987.

[389] F. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19), 2229, 1987.

[390] E. Polak. Computational methods in optimization: a unified approach. *Academic Press*, 1971.

[391] L. Polanyi and A. Zaenen. Contextual valence shifters. *Computing Attitude and Affect in Text: Theory and Applications*, pp. 1–10, Springer, 2006.

[392] G. Pollastri, D. Przybylski, B. Rost, and P. Baldi. Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles. *Proteins: Structure, Function, and Bioinformatics*, 47(2), pp. 228–235, 2002.

[393] J. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1), pp. 77–105, 1990.

[394] B. Polyak and A. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4), pp. 838–855, 1992.

[395] D. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. *Technical Report*, Carnegie Mellon University, 1989.

[396] B. Poole, J. Sohl-Dickstein, and S. Ganguli. Analyzing noise in autoencoders and deep networks. *arXiv:1406.1831*, 2014. https://arxiv.org/abs/1406.1831

[397] T. Plotz and S. Roth. Neural nearest neighbors networks. *NeurIPS*, pp. 1087–1098, 2018.

[398] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434*, 2015. https://arxiv.org/abs/1511.06434

[399] A. Radford *et al.* Language models are unsupervised multitask learners. *OpenAI blog*, 1(8), 2019.

[400] C. Raffel *et al.* Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv:1910.10683*, 2019. https://arxiv.org/abs/1910.10683

[401] A. Rahimi and B. Recht. Random features for large-scale kernel machines. *NeurIPS*, pp. 1177–1184, 2008.

[402] M.' A. Ranzato, Y-L. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. *NeurIPS*, pp. 1185–1192, 2008.

[403] M.' A. Ranzato, F. J. Huang, Y-L. Boureau, and Y. LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. *CVPR*, pp. 1–8, 2007.

[404] A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko. Semi-supervised learning with ladder networks. *NeurIPS*, pp. 3546–3554, 2015.

[405] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *ECCV*, pp. 525–542, 2016.

[406] A. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *CVPR Workshops*, pp. 806–813, 2014.

[407] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *CVPR*, pp. 779–788, 2016.

[408] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. Generative adversarial text to image synthesis. *ICML Conference*, pp. 1060–1069, 2016.

[409] S. Reed and N. de Freitas. Neural programmer-interpreters. *arXiv:1511.06279*, 2015.

[410] R. Rehurek and P. Sojka. Software framework for topic modelling with large corpora. *LREC 2010 Workshop on New Challenges for NLP Frameworks*, pp. 45–50, 2010. https://radimrehurek.com/gensim/index.html

[411] M. Ren, R. Kiros, and R. Zemel. Exploring models and data for image question answering. *NeurIPS*, pp. 2953–2961, 2015.

[412] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *NeurIPS*, 2015.

[413] S. Rendle. Factorization machines. *IEEE ICDM Conference*, pp. 995–100, 2010.

[414] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive autoencoders: Explicit invariance during feature extraction. *ICML*, pp. 833–840, 2011.

[415] S. Rifai, Y. Dauphin, P. Vincent, Y. Bengio, and X. Muller. The manifold tangent classifier. *NeurIPS*, pp. 2294–2302, 2011.

[416] D. Rezende, S. Mohamed, and D. Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *arXiv:1401.4082*, 2014. https://arxiv.org/abs/1401.4082

[417] R. Rifkin. Everything old is new again: a fresh look at historical approaches in machine learning. *Ph.D. Thesis*, Massachusetts Institute of Technology, 2002.

[418] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5, pp. 101–141, 2004.

[419] V. Romanuke. Parallel Computing Center (Khmelnitskiy, Ukraine) represents an ensemble of 5 convolutional neural networks which performs on MNIST at 0.21 percent error rate. Retrieved 24 November 2016.

[420] X. Rong. Word2vec parameter learning explained. *arXiv:1411.2738*, 2014. https://arxiv.org/abs/1411.2738

[421] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386, 1958.

[422] D. Ruck, S. Rogers, and M. Kabrisky. Feature selection using a multilayer perceptron. *Journal of Neural Network Computing*, 2(2), pp. 40–88, 1990.

[423] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE TPAMI*, 20(1), pp. 23–38, 1998.

[424] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323 (6088), pp. 533–536, 1986.

[425] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by back-propagating errors. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pp. 318–362, 1986.

[426] D. Rumelhart, D. Zipser, and J. McClelland. Parallel Distributed Processing, *MIT Press*, pp. 151–193, 1986.

[427] D. Rumelhart and D. Zipser. Feature discovery by competitive learning. *Cognitive science*, 9(1), pp. 75–112, 1985.

[428] G. Rummery, and M. Niranjan. Online Q-learning using connectionist systems (Vol. 37). *University of Cambridge, Department of Engineering*, 1994.

[429] A. M. Rush, S. Chopra, and J. Weston. A Neural Attention Model for Abstractive Sentence Summarization. *arXiv:1509.00685*, 2015. https://arxiv.org/abs/1509.00685

[430] S. Sabour, N. Frosst, and G. Hinton. Dynamic routing between capsules. *arXiv:1710.09829*, 2017. https://arxiv.org/abs/1710.09829

[431] R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted Boltzmann machines for collaborative filtering. *ICML*, pp. 791–798, 2007.

[432] R. Salakhutdinov and G. Hinton. Semantic Hashing. *SIGIR Workshop on Information Retrieval and Applications of Graphical Models*, 2007.

[433] A. Santoro *et al.* One shot learning with memory-augmented neural networks. *arXiv:1605:06065*, 2016. https://arxiv.org/abs/1605.06065

[434] R. Salakhutdinov and G. Hinton. Deep Boltzmann machines. *AISTATS*, pp. 448–455, 2009.

[435] R. Salakhutdinov and H. Larochelle. Efficient Learning of Deep Boltzmann Machines. *AISTATS*, pp. 693–700, 2010.

[436] T. Salimans and D. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *NeurIPS*, pp. 901–909, 2016.

[437] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. *NeurIPS*, pp. 2234–2242, 2016.

[438] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, pp. 210–229, 1959.

[439] T Sanger. Neural network learning control of robot manipulators using gradually increasing task difficulty. *IEEE Transactions on Robotics and Automation*, 10(3), 1994.

[440] H. Sarimveis, A. Alexandridis, and G. Bafas. A fast training algorithm for RBF networks based on subtractive clustering. *Neurocomputing*, 51, pp. 501–505, 2003.

[441] W. Saunders, G. Sastry, A. Stuhlmueller, and O. Evans. Trial without Error: Towards Safe Reinforcement Learning via Human Intervention. *arXiv:1707.05173*, 2017. https://arxiv.org/abs/1707.05173

[442] A. Saxe, P. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Ng. On random weights and unsupervised feature learning. *ICML*, pp. 1089–1096, 2011.

[443] A. Saxe, J. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*, 2013. https://arxiv.org/abs/1312.6120

[444] F. Scarselli *et al.* The graph neural network model. *IEEE TNNLS*, 20(1), pp. 61–80, 2008.

[445] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6), pp. 233–242, 1999.

[446] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv:1511.05952*, 2015. https://arxiv.org/abs/1511.05952

[447] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. *ICML*, pp. 343–351, 2013.

[448] M. Schlichtkrull *et al.* Modeling relational data with graph convolutional networks. *European Semantic Web Conference*, Springer, 2018.

[449] B. Schölkopf *et al.* Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45(11), pp. 2758–2765, 1997.

[450] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61, pp. 85–117, 2015.

[451] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. *ICML*, 2015.

[452] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *ICLR*, 2016.

[453] J. Schulman *et al.* Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. https://arxiv.org/abs/1707.06347

[454] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), pp. 2673–2681, 1997.

[455] H. Schwenk and Y. Bengio. Boosting neural networks. *Neural Computation*, 12(8), pp. 1869–1887, 2000.

[456] S. Sedhain, A. K. Menon, S. Sanner, and L. Xie. Autorec: Autoencoders meet collaborative filtering. *WWW*, pp. 111–112, 2015.

[457] T. J. Sejnowski. Higher-order Boltzmann machines. *AIP Conference Proceedings*, 15(1), pp. 298–403, 1986.

[458] G. Seni and J. Elder. Ensemble methods in data mining: Improving accuracy through combining predictions. *Morgan and Claypool*, 2010.

[459] B. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv:1508.07909*, 2015. https://arxiv.org/abs/1508.07909

[460] I. Serban, A. Sordoni, R. Lowe, L. Charlin, J. Pineau, A. Courville, and Y. Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. *AAAI*, pp. 3295–3301, 2017.

[461] I. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. *AAAI*, pp. 3776–3784, 2016.

[462] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv:1312.6229*, 2013. https://arxiv.org/abs/1312.6229

[463] J. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Technical Report, CMU-CS-94-125*, Carnegie-Mellon University, 1994.

[464] H. Siegelmann and E. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1), pp. 132–150, 1995.

[465] D. Silver *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature*, 529.7587, pp. 484–489, 2016.

[466] D. Silver *et al.* Mastering the game of go without human knowledge. *Nature*, 550.7676, pp. 354–359, 2017.

[467] D. Silver *et al.* Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*, 2017. https://arxiv.org/abs/1712.01815

[468] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127(1), pp. 3–30, 2011.

[469] E. Shelhamer, J., Long, and T. Darrell. Fully convolutional networks for semantic segmentation. *IEEE TPAMI*, 39(4), pp. 640–651, 2017.

[470] J. Sietsma and R. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1), pp. 67–79, 1991.

[471] B. W. Silverman. Density Estimation for Statistics and Data Analysis. *Chapman and Hall*, 1986.

[472] P. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. *ICDAR*, pp. 958–962, 2003.

[473] H. Simon. The Sciences of the Artificial. *MIT Press*, 1996.

[474] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014. https://arxiv.org/abs/1409.1556

[475] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. *NeurIPS*, pp. 568–584, 2014.

[476] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv:1312.6034*, 2013. https://arxiv.org/abs/1312.6034

[477] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations. pp. 194–281, 1986.

[478] J. Snoek, H. Larochelle, and R. Adams. Practical bayesian optimization of machine learning algorithms. *NeurIPS*, pp. 2951–2959, 2013.

[479] K. Sohn, H. Lee, and X. Yan. Learning structured output representation using deep conditional generative models. *NeurIPS*, 2015.

[480] R. Solomonoff. A system for incremental learning based on algorithmic probability. *Sixth Israeli Conference on Artificial Intelligence, CVPR*, pp. 515–527, 1994.

[481] Y. Song, A. Elkahky, and X. He. Multi-rate deep learning for temporal recommendation. *ACM SIGIR Conference*, pp. 909–912, 2016.

[482] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv:1412.6806*, 2014. https://arxiv.org/abs/1412.6806

[483] N. Srivastava *et al.* Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), pp. 1929–1958, 2014.

[484] N. Srivastava and R. Salakhutdinov. Multimodal learning with deep Boltzmann machines. *NeurIPS*, pp. 2222–2230, 2012.

[485] N. Srivastava, R. Salakhutdinov, and G. Hinton. Modeling documents with deep Boltzmann machines. *Uncertainty in Artificial Intelligence*, 2013.

[486] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv:1505.00387*, 2015. https://arxiv.org/abs/1505.00387

[487] A. Storkey. Increasing the capacity of a Hopfield network without sacrificing functionality. *Artificial Neural Networks*, pp. 451–456, 1997.

[488] F. Strub and J. Mary. Collaborative filtering with stacked denoising autoencoders and sparse inputs. *NeurIPS Workshop on Machine Learning for eCommerce*, 2015.

[489] S. Sukhbaatar, J. Weston, and R. Fergus. End-to-end memory networks. *NeurIPS*, pp. 2440–2448, 2015.

[490] Y. Sun, D. Liang, X. Wang, and X. Tang. Deepid3: Face recognition with very deep neural networks. *arXiv:1502.00873*, 2013. https://arxiv.org/abs/1502.00873

[491] Y. Sun, X. Wang, and X. Tang. Deep learning face representation from predicting 10,000 classes. *CVPR*, pp. 1891–1898, 2014.

[492] M. Sundermeyer, R. Schluter, and H. Ney. LSTM neural networks for language modeling. *Interspeech*, 2010.

[493] M. Sundermeyer, T. Alkhouli, J. Wuebker, and H. Ney. Translation modeling with bidirectional recurrent neural networks. *EMNLP*, pp. 14–25, 2014.

[494] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *ICML*, pp. 1139–1147, 2013.

[495] I. Sutskever and T. Tieleman. On the convergence properties of contrastive divergence. *International Conference on Artificial Intelligence and Statistics*, pp. 789–795, 2010.

[496] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *NeurIPS*, pp. 3104–3112, 2014.

[497] I. Sutskever and V. Nair. Mimicking Go experts with convolutional neural networks. *International Conference on Artificial Neural Networks*, pp. 101–110, 2008.

[498] R. Sutton. Learning to Predict by the Method of Temporal Differences, *Machine Learning*, 3, pp. 9–44, 1988.

[499] R. Sutton and A. Barto. Reinforcement Learning: An Introduction. *MIT Press*, 1998.

[500] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *NeurIPS*, pp. 1057–1063, 2000.

[501] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CVPR*, pp. 1–9, 2015.

[502] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CVPR*, pp. 2818–2826, 2016.

[503] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *AAAI*, pp. 4278–4284, 2017.

[504] X. Tang, H. Yao, Y. Sun, Y. Wang, J. Tang, C. Aggarwal, P. Mitra, and S. Wang. Investigating and mitigating degree-related biases in graph convolutional networks. *CIKM Conference*, 2020.

[505] G. Taylor, R. Fergus, Y. LeCun, and C. Bregler. Convolutional learning of spatio-temporal features. *ECCV*, pp. 140–153, 2010.

[506] G. Taylor, G. Hinton, and S. Roweis. Modeling human motion using binary latent variables. *NeurIPS*, 2006.

[507] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. *KDD*, pp. 847–855, 2013.

[508] T. Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. *ICML*, pp. 1064–1071, 2008.

[509] G. Tesauro. Practical issues in temporal difference learning. *NeurIPS*, pp. 259–266, 1992.

[510] G. Tesauro. Td-gammon: A self-teaching backgammon program. *Applications of Neural Networks*, Springer, pp. 267–285, 1992.

[511] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), pp. 58–68, 1995.

[512] Y. Teh and G. Hinton. Rate-coded restricted Boltzmann machines for face recognition. *NeurIPS*, 2001.

[513] S. Thrun. Learning to play the game of chess *NeurIPS*, pp. 1069–1076, 1995.

[514] S. Thrun and L. Platt. Learning to learn. *Springer*, 2012.

[515] Y. Tian, Q. Gong, W. Shang, Y. Wu, and L. Zitnick. ELF: An extensive, lightweight and flexible research platform for real-time strategy games. *arXiv:1707.01067*, 2017. https://arxiv.org/abs/1707.01067

[516] A. Tikhonov and V. Arsenin. Solution of ill-posed problems. *Winston and Sons*, 1977.

[517] D. Tran *et al.* Learning spatiotemporal features with 3d convolutional networks. *ICCV*, 2015.

[518] D. Tsipras *et al.* Robustness may be at odds with accuracy. *ICLR Conference*, 2019.

[519] R. Uijlings, A. van de Sande, T. Gevers, and M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2), 2013.

[520] H. Valpola. From neural PCA to deep unsupervised learning. *Advances in Independent Component Analysis and Learning Machines*, pp. 143–171, Elsevier, 2015.

[521] A. Vaswani *et al.* Attention is All you Need. *NeurIPS*, 2017. https://arxiv.org/abs/1706.03762

[522] A. Vedaldi and K. Lenc. Matconvnet: Convolutional neural networks for matlab. *ACM International Conference on Multimedia*, pp. 689–692, 2005. http://www.vlfeat.org/matconvnet/

[523] V. Veeriah, N. Zhuang, and G. Qi. Differential recurrent neural networks for action recognition. *ICCV*, pp. 4041–4049, 2015.

[524] A. Veit, M. Wilber, and S. Belongie. Residual networks behave like ensembles of relatively shallow networks. *NeurIPS*, pp. 550–558, 2016.

[525] P. Velickovic *at al.* Graph attention networks. *arXiv:1710.10903*, 2017. https://arxiv.org/abs/1710.10903

[526] P. Vincent, H. Larochelle, Y. Bengio, and P. Manzagol. Extracting and composing robust features with denoising autoencoders. *ICML*, pp. 1096–1103, 2008.

[527] O. Vinyals, C. Blundell, T. Lillicrap, and D. Wierstra. Matching networks for one-shot learning. *NeurIPS*, pp. 3530–3638, 2016.

[528] O. Vinyals and Q. Le. A Neural Conversational Model. *arXiv:1506.05869*, 2015. https://arxiv.org/abs/1506.05869

[529] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CVPR*, pp. 3156–3164, 2015.

[530] J. Walker, C. Doersch, A. Gupta, and M. Hebert. An uncertain future: Forecasting from static images using variational autoencoders. *ECCV*, pp. 835–851, 2016.

[531] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus. Regularization of neural networks using dropconnect. *ICML*, pp. 1058–1066, 2013.

[532] B. Wang *et al.* On position embeddings in BERT. *ICLR*, 2021.

[533] H. Wang, N. Wang, and D. Yeung. Collaborative deep learning for recommender systems. *KDD*, pp. 1235–1244, 2015.

[534] L. Wang, Y. Qiao, and X. Tang. Action recognition with trajectory-pooled deep-convolutional descriptors. *CVPR*, pp. 4305–4314, 2015.

[535] S. Wang, C. Aggarwal, and H. Liu. Using a random forest to inspire a neural network and improving on it. *SDM Conference*, 2017.

[536] S. Wang, C. Aggarwal, and H. Liu. Randomized feature engineering as a fast and accurate alternative to kernel methods. *KDD*, 2017.

[537] T. Wang, D. Wu, A. Coates, and A. Ng. End-to-end text recognition with convolutional neural networks. *International Conference on Pattern Recognition*, pp. 3304–3308, 2012.

[538] X. Wang and A. Gupta. Generative image modeling using style and structure adversarial networks. *ECCV*, 2016.

[539] C. J. H. Watkins. Learning from delayed rewards. *PhD Thesis*, King's College, Cambridge, 1989.

[540] C. J. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3–4), pp. 279–292, 1992.

[541] M. Welling, M. Rosen-Zvi, and G. Hinton. Exponential family harmoniums with an application to information retrieval. *NeurIPS*, pp. 1481–1488, 2005.

[542] A. Wendemuth. Learning the unlearnable. *Journal of Physics A: Math. Gen.*, 28, pp. 5423–5436, 1995.

[543] P. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. *PhD thesis, Harvard University*, 1974.

[544] P. Werbos. The roots of backpropagation: from ordered derivatives to neural networks and political forecasting (Vol. 1). *John Wiley and Sons*, 1994.

[545] P. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10), pp. 1550–1560, 1990.

[546] J. Weston, A. Bordes, S. Chopra, A. Rush, B. van Merrienboer, A. Joulin, and T. Mikolov. Towards ai-complete question answering: A set of pre-requisite toy tasks. *arXiv:1502.05698*, 2015. https://arxiv.org/abs/1502.05698

[547] J. Weston, S. Chopra, and A. Bordes. Memory networks. *ICLR*, 2015.

[548] J. Weston and C. Watkins. Multi-class support vector machines. *Technical Report CSD-TR-98-04*, Department of Computer Science, Royal Holloway, University of London, May, 1998.

[549] D. Wettschereck and T. Dietterich. Improving the performance of radial basis function networks by learning center locations. *NeurIPS*, pp. 1133–1140, 1992.

[550] B. Widrow and M. Hoff. Adaptive switching circuits. *IRE WESCON Convention Record*, 4(1), pp. 96–104, 1960.

[551] S. Wieseler and H. Ney. A convergence analysis of log-linear training. *NeurIPS*, pp. 657–665, 2011.

[552] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4), pp. 229–256, 1992.

[553] F. Wu *et al.* Simplifying graph convolutional networks. *arXiv:1902.07153* 2019. https://arxiv.org/abs/1902.07153

[554] C. Wu, A. Ahmed, A. Beutel, A. Smola, and H. Jing. Recurrent recommender networks. *WSDM Conference*, pp. 495–503, 2017.

[555] Y. Wu, C. DuBois, A. Zheng, and M. Ester. Collaborative denoising auto-encoders for top-n recommender systems. *WSDM Conference*, pp. 153–162, 2016.

[556] Y. Wu *et al.* Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144*, 2016. https://arxiv.org/abs/1609.08144

[557] Z. Wu. Global continuation for distance geometry problems. *SIAM Journal of Optimization*, 7, pp. 814–836, 1997.

[558] Z. Wu *et al.* A comprehensive survey on graph neural networks. *IEEE TNNLS*, 2020.

[559] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. *arXiv:1611.05431*, 2016. https://arxiv.org/abs/1611.05431

[560] E. Xing, R. Yan, and A. Hauptmann. Mining associated text and images with dual-wing harmoniums. *Uncertainty in Artificial Intelligence*, 2005.

[561] C. Xiong, S. Merity, and R. Socher. Dynamic memory networks for visual and textual question answering. *ICML*, pp. 2397–2406, 2016.

[562] K. Xu *et al.* Show, attend, and tell: Neural image caption generation with visual attention. *ICML*, 2015.

[563] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-gpu training of convnets. *arXiv:1312.5853*, 2013. https://arxiv.org/abs/1312.5853

[564] Z. Yang, X. He, J. Gao, L. Deng, and A. Smola. Stacked attention networks for image question answering. *CVPR*, pp. 21–29, 2016.

[565] Z. Yang *et al.* Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv:1906.08237*, 2019. https://arxiv.org/abs/1906.08237

[566] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9), pp. 1423–1447, 1999.

[567] Z. Ying *et al.* Graph convolutional neural networks for web-scale recommender systems. *KDD*, 2018.

[568] Z. Ying *et al.* Hierarchical graph representation learning with differentiable pooling. *NeurIPS*, 2018.
Code: https://github.com/RexYing/diffpool

[569] F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv:1511.07122*, 2015. https://arxiv.org/abs/1511.07122

[570] H. Yu and B. Wilamowski. Levenberg–Marquardt training. *Industrial Electronics Handbook*, 5(12), 1, 2011.

[571] L. Yu, W. Zhang, J. Wang, and Y. Yu. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. *AAAI*, pp. 2852–2858, 2017.

[572] W. Yu, W. Cheng, C. Aggarwal, K. Zhang, H. Chen, and Wei Wang. NetWalk: A flexible deep embedding approach for anomaly Detection in dynamic networks, *KDD*, 2018.

[573] W. Yu, C. Zheng, W. Cheng, C. Aggarwal, D. Song, B. Zong, H. Chen, and W. Wang. Learning deep network representations with adversarially regularized autoencoders. *KDD*, 2018.

[574] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv:1605.07146*, 2016. https://arxiv.org/abs/1605.07146

[575] W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *arXiv:1505.00521*, 2015. https://arxiv.org/abs/1505.00521

[576] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus. Learning simple algorithms from examples. *ICML*, pp. 421–429, 2016.

[577] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv:1409.2329*, 2014. https://arxiv.org/abs/1409.2329

[578] M. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv:1212.5701*, 2012. https://arxiv.org/abs/1212.5701

[579] M. Zeiler, D. Krishnan, G. Taylor, and R. Fergus. Deconvolutional networks. *CVPR*, pp. 2528–2535, 2010.

[580] M. Zeiler, G. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level feature learning. *ICCV*, pp. 2018-2025, 2011.

[581] M. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *ECCV*, Springer, pp. 818–833, 2013.

[582] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv:1611.03530*, 2016. https://arxiv.org/abs/1611.03530

[583] L. Zhang, C. Aggarwal, and G.-J. Qi. Stock Price Prediction via Discovering Multi-Frequency Trading Patterns. *KDD*, 2017.

[584] S. Zhang, L. Yao, and A. Sun. Deep learning based recommender system: A survey and new perspectives. *arXiv:1707.07435*, 2017. https://arxiv.org/abs/1707.07435

[585] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. *NeurIPS*, pp. 649–657, 2015.

[586] J. Zhao, M. Mathieu, and Y. LeCun. Energy-based generative adversarial network. *arXiv:1609.03126*, 2016. https://arxiv.org/abs/1609.03126

[587] D. Zheng, M. Wang, Q. Gan, Z. Zhang, and G. Karypis. Learning Graph Neural Networks with Deep Graph Library, *WWW Tutorial*, 2020. https://youtu.be/bD6S3xUXNds

[588] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103*, 2017. https://arxiv.org/abs/1709.00103

[589] C. Zhou and R. Paffenroth. Anomaly detection with robust deep autoencoders. *KDD*, pp. 665–674, 2017.

[590] M. Zhou, Z. Ding, J. Tang, and D. Yin. Micro Behaviors: A new perspective in e-commerce recommender systems. *WSDM Conference*, 2018.

[591] Z.-H. Zhou. Ensemble methods: Foundations and algorithms. *CRC Press*, 2012.

[592] Z.-H. Zhou, J. Wu, and W. Tang. Ensembling neural networks: many could be better than all. *Artificial Intelligence*, 137(1–2), pp. 239–263, 2002.

[593] Y. Zhu *et al.* Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *CVPR*, 2019.

[594] C. Zitnick and P. Dollar. Edge Boxes: Locating object proposals from edges. *ECCV*, pp. 391–405, 2014.

[595] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016. https://arxiv.org/abs/1611.01578

[596] http://caffe.berkeleyvision.org/

[597] http://torch.ch/

[598] https://pypi.org/project/Theano/

[599] https://www.tensorflow.org/

[600] https://keras.io/

[601] https://lasagne.readthedocs.io/en/latest/

[602] http://www.netflixprize.com/community/topic_1537.html

[603] https://arxiv.org/abs/1609.08144

[604] https://github.com/karpathy/char-rnn

[605] http://www.image-net.org/

[606] http://www.image-net.org/challenges/LSVRC/

[607] https://www.cs.toronto.edu/~kriz/cifar.html

[608] http://code.google.com/p/cuda-convnet/

[609] http://caffe.berkeleyvision.org/gathered/examples/feature_extraction.html

[610] https://github.com/caffe2/caffe2/wiki/Model-Zoo

[611] http://scikit-learn.org/

[612] http://clic.cimec.unitn.it/composes/toolkit/

[613] https://github.com/stanfordnlp/GloVe

[614] https://deeplearning4j.org/

[615] https://code.google.com/archive/p/word2vec/

[616] https://www.tensorflow.org/tutorials/word2vec/

[617] https://github.com/aditya-grover/node2vec

[618] https://github.com/caglar/autoencoders

[619] https://github.com/y0ast

[620] https://github.com/fastforwardlabs/vae-tf/tree/master

[621] https://science.education.nih.gov/supplements/webversions/BrainAddiction/guide/lesson2-1.html

[622] https://www.ibm.com/us-en/marketplace/deep-learning-platform

[623] https://www.youtube.com/watch?v=2fRnHVVLf1Y&list=PLiPvV5TNogxKKwvKb-1RKwkq2hm7ZvpHz0

[624] https://www.coursera.org/learn/neural-networks-deep-learning

[625] https://archive.ics.uci.edu/ml/datasets.html

[626] http://www.bbc.com/news/technology-35785875

[627] https://deepmind.com/blog/exploring-mysteries-alphago/

[628] http://deeplearning.mit.edu/

[629] http://karpathy.github.io/2016/05/31/rl/

[630] https://github.com/hughperkins/kgsgo-dataset-preprocessor

[631] https://www.wired.com/2016/03/two-moves-alphago-lee-sedol-redefined- future/

[632] https://qz.com/639952/googles-ai-won-the-game-go-by-defying-millennia-of-basic-human-instinct/

[633] http://www.mujoco.org/

[634] https://sites.google.com/site/gaepapersupp/home

[635] https://www.youtube.com/watch?v=1L0TKZQcUtA&list=PLrAXtmErZgOeiKm4-sgNOknGvNjby9efdf

[636] https://openai.com/

[637] http://jaberg.github.io/hyperopt/

[638] http://www.cs.ubc.ca/labs/beta/Projects/SMAC/

[639] https://github.com/JasperSnoek/spearmint

[640] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[641] https://www.youtube.com/watch?v=2pWv7GOvuf0

[642] https://gym.openai.com

[643] https://universe.openai.com

[644] https://github.com/facebookresearch/ParlAI

[645] https://github.com/openai/baselines

[646] https://github.com/carpedm20/deep-rl-tensorflow

[647] https://github.com/matthiasplappert/keras-rl

[648] http://apollo.auto/

[649] https://github.com/Element-Research/rnn/blob/master/examples/

[650] https://github.com/lmthang/nmt.matlab

[651] https://github.com/carpedm20/NTM-tensorflow

[652] https://github.com/camigord/Neural-Turing-Machine

[653] https://github.com/SigmaQuan/NTM-Keras

[654] https://github.com/snipsco/ntm-lasagne

[655] https://github.com/kaishengtai/torch-ntm

[656] https://github.com/facebook/MemNN

[657] https://github.com/carpedm20/MemN2N-tensorflow

[658] https://github.com/YerevaNN/Dynamic-memory-networks-in-Theano

[659] https://github.com/carpedm20/DCGAN-tensorflow

[660] https://github.com/carpedm20

[661] https://github.com/jacobgil/keras-dcgan

[662] https://github.com/wiseodd/generative-models

[663] https://github.com/paarthneekhara/text-to-image

[664] https://developer.nvidia.com/cudnn

[665] http://www.nvidia.com/object/machine-learning.html

[666] https://developer.nvidia.com/deep-learning-frameworks

[667] http://snap.stanford.edu

[668] http://snap.stanford.edu/graphsage/

[669] https://github.com/microsoft/tf-gnn-samples

[670] https://www.dgl.ai/

[671] https://github.com/rusty1s/pytorch_geometric

[672] https://msturing.org/

[673] https://www.tensorflow.org/tutorials/text/transformer

[674] https://github.com/google-research/bert

[675] https://github.com/google-research/text-to-text-transfer-transformer

[676] https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/transformer/moe.py

[677] https://github.com/zihangdai/xlnet

[678] https://github.com/allenai/bilm-tf

[679] https://adversarial-ml-tutorial.org/

[680] https://github.com/huggingface/transformers

[681] https://github.com/google/trax

[682] https://chat.openai.com/chatgpt

Index

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-031-29642-0