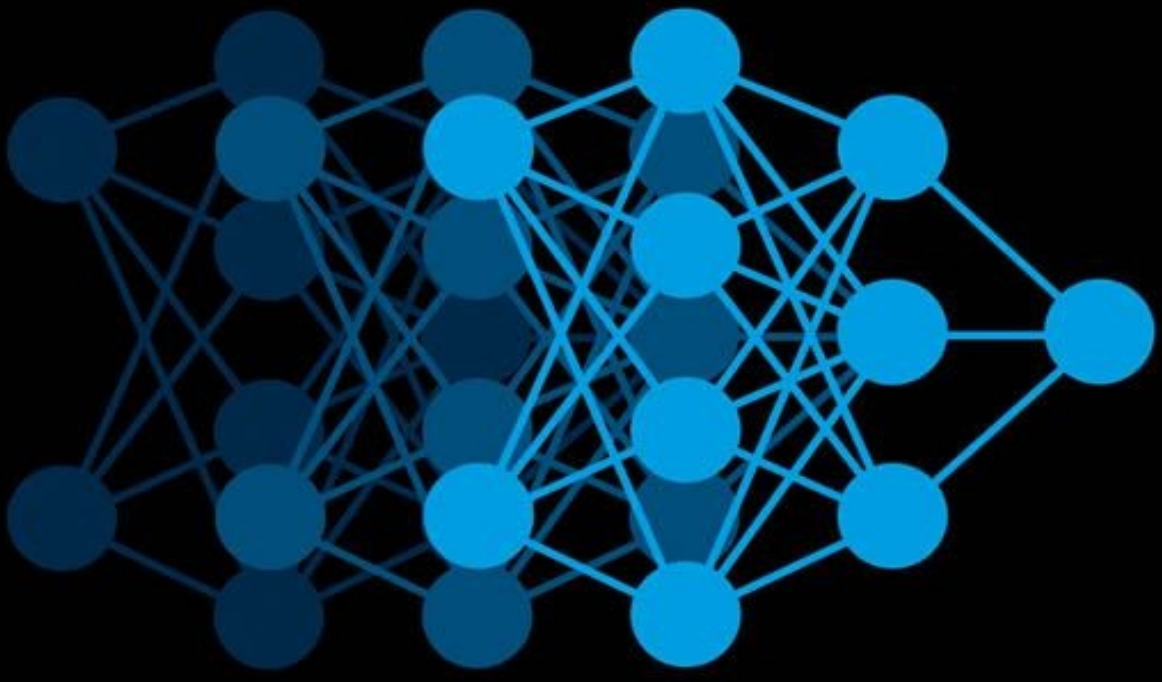
DEEP LEARNING

JOHN D. KELLEHER



THE MIT PRESS ESSENTIAL KNOWLEDGE SERIES

Deep Learning

The MIT Press Essential Knowledge Series

A complete list of the titles in this series appears at the back of this book.

Deep Learning

John D. Kelleher

The MIT Press | Cambridge, Massachusetts | London, England

This book was set in Chaparral Pro by Toppan Best-set Premedia Limited. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Kelleher, John D., 1974- author.

Title: Deep learning / John D. Kelleher.

Description: Cambridge, MA : The MIT Press, [2019] | Series: The MIT press essential knowledge series | Includes bibliographical references and index.

Identifiers: LCCN 2018059550 | ISBN 9780262537551 (pbk. : alk. paper)

Subjects: LCSH: Machine learning. | Artificial intelligence.

Classification: LCC Q325.5 .K454 2019 | DDC 006.3/1—dc23 LC record available at https://lccn.loc.gov/2018059550

10 9 8 7 6 5 4 3 2 1

Contents

Series Foreword

The MIT Press Essential Knowledge series offers accessible, concise, beautifully produced pocket-size books on topics of current interest. Written by leading thinkers, the books in this series deliver expert overviews of subjects that range from the cultural and the historical to the scientific and the technical.

In today's era of instant information gratification, we have ready access to opinions, rationalizations, and superficial descriptions. Much harder to come by is the foundational knowledge that informs a principled understanding of the world. Essential Knowledge books fill that need. Synthesizing specialized subject matter for nonspecialists and engaging critical topics through fundamentals, each of these compact volumes offers readers a point of access to complex ideas.

Bruce Tidor
Professor of Biological Engineering and Computer Science
Massachusetts Institute of Technology

Preface

Deep learning is enabling innovation and change across all aspects of our modern lives. Most of the artificial intelligence breakthroughs that you hear about in the media are based on deep learning. As a result, whether you are a business person interested in improving the efficiency of your organization, a policymaker concerned with ethics and privacy in a Big Data world, a researcher working with complex data, or a curious citizen who wants a better sense of the potential of artificial intelligence and how it will change your life, it is important for you to have an understanding of deep learning.

The goal of this book is to enable the general reader to gain an understanding of what deep learning is, where it has come from, how it works, what it makes possible (and what it doesn't), and how the field is likely to develop in the next ten years. The fact that deep learning is a set of algorithms and models means that understanding deep learning requires understanding how these algorithms and models process data. As a result, this book is not purely descriptive and definitional; it also includes explanations of algorithms. I have attempted to present the technical material in an accessible way. From my teaching experience, I have found that for technical topics the most accessible presentation is to explain the fundamental concepts in a step-by-step manner. So, although I have tried to keep the mathematical content to a minimum, where I felt it was necessary to include it I have endeavored to walk you through the mathematical equations in as clear and direct a manner as I can. I have supplemented these explanations with examples and illustrations.

What is really wondrous about deep learning is not the complexity of the math it is built on, but rather, that it can perform such a diverse set of exciting and impressive tasks using such simple calculations. Don't be surprised to find yourself saying: "Is that all it's doing?" In fact, a deep learning model really is just a lot (admittedly, an awful lot) of multiplications and additions with a few nonlinear mappings (which I will explain) added in. Yet, despite this simplicity, these models can, among other achievements, beat the Go world champion, define the state-of-the-art in computer vision and machine translation, and drive a car. This book is an introductory text about deep learning, but I hope that it is an introduction that has enough depth that you will come back to the book as your confidence with the material grows.

Acknowledgments

This book would not have been possible without the sacrifices made by my wife, Aphra, and my family, in particular my parents John and Betty Kelleher. I have also received a huge amount of support from friends, especially Alan McDonnell, Ionela Lungu, Simon Dobnik, Lorraine Byrne, Noel Fitzpatrick, and Josef van Genabith.

I would also like to acknowledge the help I have received from the staff at the MIT Press, and from a number of people who have read sections of the book and provided feedback. MIT Press organized three anonymous reviewers who read and commented on a draft of the book. I thank these reviewers for their time and helpful feedback. Also a number of people read draft chapters from the book and I wish to take this opportunity to publicly acknowledge their help, so my thanks to: Mike Dillinger, Magdalena Kacmajor, Elizabeth Kelleher, John Bernard Kelleher, Aphra Kerr, Filip Klubička, and Abhijit Mahalunkar. This book has been informed by the many conversations I have had with colleagues and students about deep learning, in particular those with Robert Ross and Giancarlo Salton.

This book is dedicated to my sister Elizabeth (Liz) Kelleher in recognition of her love and support, and her patience with a brother who can’t stop explaining things.

1

Introduction to Deep Learning

Deep learning is the subfield of artificial intelligence that focuses on creating large neural network models that are capable of making accurate *data-driven decisions*. Deep learning is particularly suited to contexts where the data is complex and where there are large datasets available. Today most online companies and high-end consumer technologies use deep learning. Among other things, Facebook uses deep learning to analyze text in online conversations. Google, Baidu, and Microsoft all use deep learning for image search, and also for machine translation. All modern smart phones have deep learning systems running on them; for example, deep learning is now the standard technology for speech recognition, and also for face detection on digital cameras. In the healthcare sector, deep learning is used to process medical images (X-rays, CT, and MRI scans) and diagnose health conditions. Deep learning is also at the core of self-driving cars, where it is used for localization and mapping, motion planning and steering, and environment perception, as well as tracking driver state.

Perhaps the best-known example of deep learning is DeepMind's AlphaGo.[1] Go is a board game similar to Chess. AlphaGo was the first computer program to beat a professional Go player. In March 2016, it beat the top Korean professional, Lee Sedol, in a match watched by more than two hundred million people. The following year, in 2017, AlphaGo beat the world's No. 1 ranking player, China's Ke Jie.

In 2016 AlphaGo's success was very surprising. At the time, most people expected that it would take many more years of research before a computer would be able to compete with top level human Go players. It had been known for a long time that programming a computer to play Go was much more difficult than programming it to play Chess. There are many more board configurations possible in Go than there are in Chess. This is because Go has a larger board and simpler rules than Chess. There are, in fact, more possible board configurations in Go than there are atoms in the universe. This massive search space and Go's large branching factor (the number of board configurations that can be reached in one move) makes Go an incredibly challenging game for both humans and computers.

One way of illustrating the relative difficulty Go and Chess presented to computer programs is through a historical comparison of how Go and Chess programs competed with human players. In 1967, MIT's MacHack-6 Chess program could successfully compete with humans and had an Elo rating[2] well above novice level, and, by May 1997, DeepBlue was capable of beating the Chess world champion Gary Kasparov. In comparison, the first complete Go program wasn't written until 1968 and strong human players were still able to easily beat the best Go programs in 1997.

The time lag between the development of Chess and Go computer programs reflects the difference in computational difficulty between these two games. However, a second historic comparison between Chess and Go illustrates the revolutionary impact that deep learning has had on the ability of computer programs to compete with humans at Go. It took thirty years for Chess programs to progress from human level competence in 1967 to world champion level in 1997. However, with the development of deep learning it took only seven years for computer Go programs to progress from advanced amateur to world champion; as recently as 2009 the best Go program in the world was rated at the low-end of advanced amateur. This acceleration in performance through the use of deep learning is nothing short of extraordinary, but it is also indicative of the types of progress that deep learning has enabled in a number of fields.

AlphaGo uses deep learning to evaluate board configurations and to decide on the next move to make. The fact that AlphaGo used deep learning to decide what move to make next is a clue to understanding why deep learning is useful across so many different domains and applications. Decision-making is a crucial part of life. One way to make decisions is to base them on your "intuition" or your "gut feeling." However, most people would agree that the best way to make decisions is to base them on the relevant data. Deep learning enables *data-driven decisions* by identifying and extracting patterns from large datasets that accurately map from sets of complex inputs to good decision outcomes.

Artificial Intelligence, Machine Learning, and Deep Learning

Deep learning has emerged from research in artificial intelligence and machine learning. Figure 1.1 illustrates the relationship between artificial intelligence, machine learning, and deep learning.

Deep learning enables *data-driven decisions* by identifying and extracting patterns from large datasets that accurately map from sets of complex inputs to good decision outcomes.

The field of artificial intelligence was born at a workshop at Dartmouth College in the summer of 1956. Research on a number of topics was presented at the workshop including mathematical theorem proving, natural language processing, planning for games, computer programs that could learn from examples, and neural networks. The modern field of machine learning draws on the last two topics: computers that could learn from examples, and neural network research.

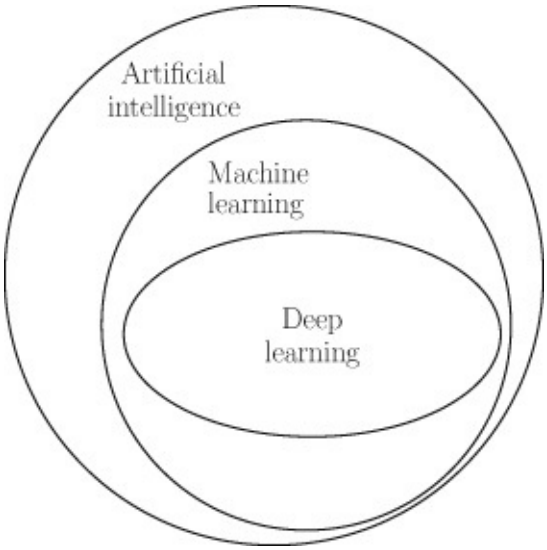


Figure 1.1 The relationship between artificial intelligence, machine learning, and deep learning.

Machine learning involves the development and evaluation of algorithms that enable a computer to extract (or learn) functions from a dataset (sets of examples). To understand what machine learning means we need to understand three terms: dataset, algorithm, and function.

In its simplest form, a dataset is a table where each row contains the description of one example from a domain, and each column contains the information for one of the features in a domain. For example, table 1.1 illustrates an example dataset for a loan application domain. This dataset lists the details of four example loan applications. Excluding the ID feature, which is only for ease of reference, each example is described using three features: the applicant's annual income, their current debt, and their credit solvency.

Table 1.1. A dataset of loan applicants and their known credit solvency ratings

ID	Annual Income	Current Debt	Credit Solvency
1	$150	-$100	100
2	$250	-$300	-50
3	$450	-$250	400
4	$200	-$350	-300

An algorithm is a process (or recipe, or program) that a computer can follow. In the context of machine learning, an algorithm defines a process to analyze a dataset and identify recurring patterns in the data. For example, the algorithm might find a pattern that relates a person's annual income and current debt to their credit solvency rating. In mathematics, relationships of this type are referred to as functions.

A function is a deterministic mapping from a set of input values to one or more output values. The fact that the mapping is deterministic means that for any specific set of inputs a function will always return the same outputs. For example, addition is a deterministic mapping, and so 2+2 is always equal to 4. As we will discuss later, we can create functions for domains that are more complex than basic arithmetic, we can for example define a function that takes a person's income and debt as inputs and returns their credit solvency rating as the output value. The concept of a function is very important to deep learning so it is worth repeating the definition for emphasis: a function is simply a mapping from

inputs to outputs. In fact, the goal of machine learning is to learn functions from data. A function can be represented in many different ways: it can be as simple as an arithmetic operation (e.g., addition or subtraction are both functions that take inputs and return a single output), a sequence of *if-then-else* rules, or it can have a much more complex representation.

A function is a deterministic mapping from a set of input values to one or more output values.

One way to represent a function is to use a neural network. Deep learning is the subfield of machine learning that focuses on deep neural network models. In fact, the patterns that deep learning algorithms extract from datasets are functions that are represented as neural networks. Figure 1.2 illustrates the structure of a neural network. The boxes on the left of the figure represent the memory locations where inputs are presented to the network. Each of the circles in this figure is called a neuron and each neuron implements a function: it takes a number of values as input and maps them to an output value. The arrows in the network show how the outputs of each neuron are passed as inputs to other neurons. In this network, information flows from left to right. For example, if this network were trained to predict a person's credit solvency, based on their income and debt, it would receive the income and debt as inputs on the left of the network and output the credit solvency score through the neuron on the right.

A neural network uses a divide-and-conquer strategy to learn a function: each neuron in the network learns a simple function, and the overall (more complex) function, defined by the network, is created by combining these simpler functions. Chapter 3 will describe how a neural network processes information.

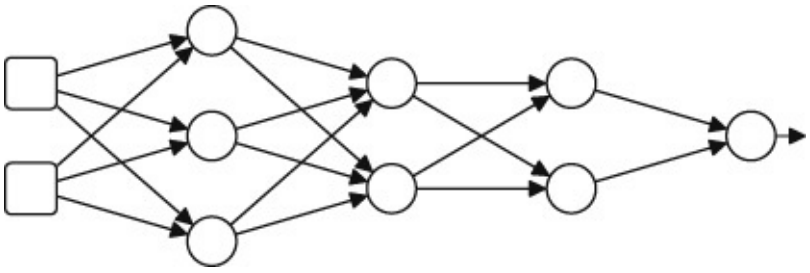


Figure 1.2 Schematic illustration of a neural network.

What Is Machine Learning?

A machine learning algorithm is a search process designed to choose the best function, from a set of possible functions, to explain the relationships between features in a dataset. To get an intuitive understanding of what is involved in extracting, or learning, a function from data, examine the following set of sample inputs to an unknown function and the outputs it returns. Given these examples, decide which arithmetic operation (addition, subtraction, multiplication, or division) is the best choice to explain the mapping the unknown function defines between its inputs and output:

$$\text{function}(\text{Inputs}) = \text{Output}$$
$$\text{function}(5,5) = 25$$
$$\text{function}(2,6) = 12$$
$$\text{function}(4,4) = 16$$
$$\text{function}(2,2) = 04$$

Most people would agree that multiplication is the best choice because it provides the best match to the observed relationship, or mapping, from the inputs to the outputs:

$$5 \times 5 = 25$$
$$2 \times 6 = 12$$
$$4 \times 7 = 28$$
$$2 \times 2 = 04$$

In this particular instance, choosing the best function is relatively straightforward, and a human can do it without the aid of a computer. However, as the number of inputs to the unknown function increases (perhaps to hundreds or thousands of inputs), and the variety of potential functions to be considered gets larger, the task becomes much more difficult. It is in these contexts that harnessing the power of machine learning to search for the best function, to match the patterns in the dataset, becomes necessary.

Machine learning involves a two-step process: training and inference. During training, a machine learning algorithm processes a dataset and chooses the

function that best matches the patterns in the data. The extracted function will be encoded in a computer program in a particular form (such as if-then-else rules or parameters of a specified equation). The encoded function is known as a model, and the analysis of the data in order to extract the function is often referred to as training the model. Essentially, models are functions encoded as computer programs. However, in machine learning the concepts of function and model are so closely related that the distinction is often skipped over and the terms may even be used interchangeably.

In the context of deep learning, the relationship between functions and models is that the function extracted from a dataset during training is represented as a neural network model, and conversely a neural network model encodes a function as a computer program. The standard process used to train a neural network is to begin training with a neural network where the parameters of the network are randomly initialized (we will explain network parameters later; for now just think of them as values that control how the function the network encodes works). This randomly initialized network will be very inaccurate in terms of its ability to match the relationship between the various input values and target outputs for the examples in the dataset. The training process then proceeds by iterating through the examples in the dataset, and, for each example, presenting the input values to the network and then using the difference between the output returned by the network and the correct output for the example listed in the dataset to update the network's parameters so that it matches the data more closely. Once the machine learning algorithm has found a function that is sufficiently accurate (in terms of the outputs it generates matching the correct outputs listed in the dataset) for the problem we are trying to solve, the training process is completed, and the final model is returned by the algorithm. This is the point at which the learning in machine learning stops.

Once training has finished, the model is fixed. The second stage in machine learning is inference. This is when the model is applied to new examples—examples for which we do not know the correct output value, and therefore we want the model to generate estimates of this value for us. Most of the work in machine learning is focused on how to train accurate models (i.e., extracting an accurate function from data). This is because the skills and methods required to deploy a trained machine learning model into production, in order to do inference on new examples at scale, are different from those that a typical data scientist will possess. There is a growing recognition within the industry of the distinctive skills needed to deploy artificial intelligence systems at scale, and this is reflected in a growing interest in the field known as DevOps, a term

describing the need for collaboration between development and operations teams (the operations team being the team responsible for deploying a developed system into production and ensuring that these systems are stable and scalable). The terms MLOps, for machine learning operations, and AIOps, for artificial intelligence operations, are also used to describe the challenges of deploying a trained model. The questions around model deployment are beyond the scope of this book, so we will instead focus on describing what deep learning is, what it can be used for, how it has evolved, and how we can train accurate deep learning models.

One relevant question here is: why is extracting a function from data useful? The reason is that once a function has been extracted from a dataset it can be applied to unseen data, and the values returned by the function in response to these new inputs can provide insight into the correct decisions for these new problems (i.e., it can be used for inference). Recall that a function is simply a deterministic mapping from inputs to outputs. The simplicity of this definition, however, hides the variety that exists within the set of functions. Consider the following examples:

- Spam filtering is a function that takes an email as input and returns a value that classifies the email as spam (or not).
- Face recognition is a function that takes an image as input and returns a labeling of the pixels in the image that demarcates the face in the image.
- Gene prediction is a function that takes a genomic DNA sequence as input and returns the regions of the DNA that encode a gene.
- Speech recognition is a function that takes an audio speech signal as input and returns a textual transcription of the speech.
- Machine translation is a function that takes a sentence in one language as input and returns the translation of that sentence in another language.

It is because the solutions to so many problems across so many domains can be framed as functions that machine learning has become so important in recent years.

Why Is Machine Learning Difficult?

There are a number of factors that make the machine learning task difficult, even with the help of a computer. First, most datasets will include noise[3] in the data, so searching for a function that matches the data exactly is not necessarily the best strategy to follow, as it is equivalent to learning the noise. Second, it is often the case that the set of possible functions is larger than the set of examples in the dataset. This means that machine learning is an ill-posed problem: the information given in the problem is not sufficient to find a *single* best solution; instead multiple possible solutions will match the data. We can use the problem of selecting the arithmetic operation (addition, subtraction, multiplication, or division) that best matches a set of example input-output mappings for an unknown function to illustrate the concept of an ill-posed problem. Here are the example mappings for this function selection problem:

$$\text{function}(\text{Inputs}) = \text{Output}$$
$$\text{function}(1,1) = 1$$
$$\text{function}(2,1) = 2$$
$$\text{function}(3,1) = 3$$

Given these examples, multiplication and division are better matches for the unknown function than addition and subtraction. However, it is not possible to decide whether the unknown function is actually multiplication or division using this sample of data, because both operations are consistent with all the examples provided. Consequently, this is an ill-posed problem: it is not possible to select a single best answer given the information provided in the problem.

One strategy to solve an ill-posed problem is to collect more data (more examples) in the hope that the new examples will help us to discriminate between the correct underlying function and the remaining alternatives. Frequently, however, this strategy is not feasible, either because the extra data is not available or is too expensive to collect. Instead, machine learning algorithms overcome the ill-posed nature of the machine learning task by supplementing the information provided by the data with a set of assumptions about the characteristics of the best function, and use these assumptions to influence the process used by the algorithm that selects the best function (or model). These assumptions are known as the inductive bias of the algorithm because in logic a

process that infers a general rule from a set of specific examples is known as inductive reasoning. For example, if all the swans that you have seen in your life are white, you might induce from these examples the general rule that *all swans are white*. This concept of inductive reasoning relates to machine learning because a machine learning algorithm induces (or extracts) a general rule (a function) from a set of specific examples (the dataset). Consequently, the assumptions that bias a machine learning algorithm are, in effect, biasing an inductive reasoning process, and this is why they are known as the inductive bias of the algorithm.

So, a machine learning algorithm uses two sources of information to select the best function: one is the dataset, and the other (the inductive bias) is the assumptions that bias the algorithm to prefer some functions over others, irrespective of the patterns in the dataset. The inductive bias of a machine learning algorithm can be understood as providing the algorithm with a perspective on a dataset. However, just as in the real world, where there is no single best perspective that works in all situations, there is no single best inductive bias that works well for all datasets. This is why there are so many different machine learning algorithms: each algorithm encodes a different inductive bias. The assumptions encoded in the design of a machine leanring algorithm can vary in strength. The stronger the assumptions the less freedom the algorithm is given in selecting a function that fits the patterns in the dataset. In a sense, the dataset and inductive bias counterbalance each other: machine learning algorithms that have a strong inductive bias pay less attention to the dataset when selecting a function. For example, if a machine learning algorithm is coded to prefer a very simple function, no matter how complex the patterns in the data, then it has a very strong inductive bias.

In chapter 2 we will explain how we can use the equation of a line as a template structure to define a function. The equation of the line is a very simple type of mathematical function. Machine learning algorithms that use the equation of a line as the template structure for the functions they fit to a dataset make the assumption that the model they generate should encode a simple linear mapping from inputs to output. This assumption is an example of an inductive bias. It is, in fact, an example of a strong inductive bias, as no matter how complex (or nonlinear) the patterns in the data are the algorithm will be restricted (or biased) to fit a linear model to it.

One of two things can go wrong if we choose a machine learning algorithm with the wrong bias. First, if the inductive bias of a machine learning algorithm is too strong, then the algorithm will ignore important information in the data

and the returned function will not capture the nuances of the true patterns in the data. In other words, the returned function will be too simple for the domain,[4] and the outputs it generates will not be accurate. This outcome is known as the function underfitting the data. Alternatively, if the bias is too weak (or permissive), the algorithm is allowed too much freedom to find a function that closely fits the data. In this case, the returned function is likely to be too complex for the domain, and, more problematically, the function is likely to fit to the noise in the sample of the data that was supplied to the algorithm during training. Fitting to the noise in the training data will reduce the function's ability to generalize to new data (data that is not in the training sample). This outcome is known as overfitting the data. Finding a machine learning algorithm that balances data and inductive bias appropriately for a given domain is the key to learning a function that neither underfits or overfits the data, and that, therefore, generalizes successfully in that domain (i.e., that is accurate at inference, or processing new examples that were not in the training data).

However, in domains that are complex enough to warrant the use of machine learning, it is not possible in advance to know what are the correct assumptions to use to bias the selection of the correct model from the data. Consequently, data scientists must use their intuition (i.e., make informed guesses) and also use trial-and-error experimentation in order to find the best machine learning algorithm to use in a given domain.

Neural networks have a relatively weak inductive bias. As a result, generally, the danger with deep learning is that the neural network model will overfit, rather than underfit, the data. It is because neural networks pay so much attention to the data that they are best suited to contexts where there are very large datasets. The larger the dataset, the more information the data provides, and therefore it becomes more sensible to pay more attention to the data. Indeed, one of the most important factors driving the emergence of deep learning over the last decade has been the emergence of Big Data. The massive datasets that have become available through online social platforms and the proliferation of sensors have combined to provide the data necessary to train neural network models to support new applications in a range of domains. To give a sense of the scale of the big data used in deep learning research, Facebook's face recognition software, DeepFace, was trained on a dataset of four million facial images belonging to more than four thousand identities (Taigman et al. 2014).

The Key Ingredients of Machine Learning

The above example of deciding which arithmetic operation best explains the relationship between inputs and outputs in a set of data illustrates the three key ingredients in machine learning:

1. Data (a set of historical examples).
2. A set of functions that the algorithm will search through to find the best match with the data.
3. Some measure of fitness that can be used to evaluate how well each candidate function matches the data.

All three of these ingredients must be correct if a machine learning project is to succeed; below we describe each of these ingredients in more detail.

We have already introduced the concept of a dataset as a two-dimensional table (or $n \times m$ matrix),[5] where each row contains the information for one example, and each column contains the information for one of the features in the domain. For example, table 1.2 illustrates how the sample inputs and outputs of the first unknown arithmetic function problem in the chapter can be represented as a dataset. This dataset contains four examples (also known as instances), and each example is represented using two input features and one output (or target) feature. Designing and selecting the features to represent the examples is a very important step in any machine learning project.

As is so often the case in computer science, and machine learning, there is a tradeoff in feature selection. If we choose to include only a minimal number of features in the dataset, then it is likely that a very informative feature will be excluded from the data, and the function returned by the machine learning algorithm will not work well. Conversely, if we choose to include as many features as possible in the domain, then it is likely that irrelevant or redundant features will be included, and this will also likely result in the function not working well. One reason for this is that the more redundant or irrelevant features that are included, the greater the probability for the machine learning algorithm to extract patterns that are based on spurious correlations between these features. In these cases, the algorithm gets confused between the real patterns in the data and the spurious patterns that only appear in the data due to the particular sample of examples that have been included in the dataset.

Finding the correct set of features to include in a dataset involves engaging

with experts who understand the domain, using statistical analysis of the distribution of individual features and also the correlations between pairs of features, and a trial-and-error process of building models and checking the performance of the models when particular features are included or excluded. This process of dataset design is a labor-intensive task that often takes up a significant portion of the time and effort expended on a machine learning project. It is, however, a critical task if the project is to succeed. Indeed, identifying which features are informative for a given task is frequently where the real value of machine learning projects emerge.

The second ingredient in a machine learning project is the set of candidate functions that the algorithm will consider as the potential explanation of the patterns in the data. In the unknown arithmetic function scenario previously given, the set of considered functions was explicitly specified and restricted to four: *addition*, *subtraction*, *multiplication*, or *division*. More generally, the set of functions is implicitly defined through the inductive bias of the machine learning algorithm and the function representation (or model) that is being used. For example, a neural network model is a very flexible function representation.

Table 1.2. A simple tabular dataset

Input 1	Input 2	Target
5	5	25
2	6	12
4	4	16
2	2	04

The third and final ingredient to machine learning is the measure of fitness. The measure of fitness is a function that takes the outputs from a candidate function, generated when the machine learning algorithm applies the candidate function to the data, and compares these outputs with the data, in some way. The result of this comparison is a value that describes the fitness of the candidate function relative to the data. A fitness function that would work for our unknown

arithmetic function scenario is to count in how many of the examples a candidate function returns a value that exactly matches the target specified in the data. Multiplication would score four out of four on this fitness measure, addition would score one out of four, and division and subtraction would both score zero out of four. There are a large variety of fitness functions that can be used in machine learning, and the selection of the correct fitness function is crucial to the success of a machine learning project. The design of new fitness functions is a rich area of research in machine learning. Varying how the dataset is represented, and how the candidate functions and the fitness function are defined, results in three different categories of machine learning: supervised, unsupervised, and reinforcement learning.

Supervised, Unsupervised, and Reinforcement Learning

Supervised machine learning is the most common type of machine learning. In supervised machine learning, each example in the dataset is labeled with the expected output (or target) value. For example, if we were using the dataset in table 1.1 to learn a function that maps from the inputs of annual income and debt to a credit solvency score, the credit solvency feature in the dataset would be the target feature. In order to use supervised machine learning, our dataset must list the value of the target feature for every example in the dataset. These target feature values can sometimes be very difficult, and expensive, to collect. In some cases, we must pay human experts to label each example in a dataset with the correct target value. However, the benefit of having these target values in the dataset is that the machine learning algorithm can use these values to help the learning process. It does this by comparing the outputs a function produces with the target outputs specified in the dataset, and using the difference (or error) to evaluate the fitness of the candidate function, and use the fitness evaluation to guide the search for the best function. It is because of this feedback from the target labels in the dataset to the algorithm that this type of machine learning is considered supervised. This is the type of machine learning that was demonstrated by the example of choosing between different arithmetic functions to explain the behavior of an unknown function.

Unsupervised machine learning is generally used for clustering data. For example, this type of data analysis is useful for customer segmentation, where a company wishes to segment its customer base into coherent groups so that it can target marketing campaigns and/or product designs to each group. In unsupervised machine learning, there are no target values in the dataset. Consequently, the algorithm cannot directly evaluate the fitness of a candidate function against the target values in the dataset. Instead, the machine learning algorithm tries to identify functions that map similar examples into clusters, such that the examples in a cluster are more similar to the other examples in the same cluster than they are to examples in other clusters. Note that the clusters are not prespecified, or at most they are initially very underspecified. For example, the data scientist might provide the algorithm with a target number of clusters, based on some intuition about the domain, without providing explicit information on relative sizes of the clusters or regarding the characteristics of examples that belong in each cluster. Unsupervised machine learning algorithms often begin by guessing an initial clustering of the examples and then iteratively adjusting the

clusters (by dropping instances from one cluster and adding them to another) so as to improve the fitness of the cluster set. The fitness functions used in unsupervised machine learning generally reward candidate functions that result in higher similarity within individual clusters and, also, high diversity between clusters.

Reinforcement learning is most relevant for online control tasks, such as robot control and game playing. In these scenarios, an agent needs to learn a policy for how it should act in an environment in order to be rewarded. In reinforcement learning, the goal of the agent is to learn a mapping from its current observation of the environment and its own internal state (its memory) to what action it should take: for instance, *should the robot move forward or backward* or *should the computer program move the pawn or take the queen*. The output of this policy (function) is the action that the agent should take next, given the current context. In these types of scenarios, it is difficult to create historic datasets, and so reinforcement learning is often carried out *in situ*: an agent is released into an environment where it experiments with different policies (starting with a potentially random policy) and over time updates its policy in response to the rewards it receives from the environment. If an action results in a positive reward, the mapping from the relevant observations and state to that action is reinforced in the policy, whereas if an action results in a negative reward, the mapping is weakened. Unlike in supervised and unsupervised machine learning, in reinforcement learning, the fact that learning is done in situ means that the training and inference stages are interleaved and ongoing. The agent infers what action it should do next and uses the feedback from the environment to learn how to update its policy. A distinctive aspect of reinforcement learning is that the target output of the learned function (the agent's actions) is decoupled from the reward mechanism. The reward may be dependent on multiple actions and there may be no reward feedback, either positive or negative, available directly after an action has been performed. For example, in a chess scenario, the reward may be +1 if the agent wins the game and -1 if the agent loses. However, this reward feedback will not be available until the last move of the game has been completed. So, one of the challenges in reinforcement learning is designing training mechanisms that can distribute the reward appropriately back through a sequence of actions so that the policy can be updated appropriately. Google's DeepMind Technologies generated a lot of interest by demonstrating how reinforcement learning could be used to train a deep learning model to learn control policies for seven different Atari computer games (Mnih et al. 2013). The input to the system was the raw pixel values from the screen, and the control

policies specified what joystick action the agent should take at each point in the game. Computer game environments are particularly suited to reinforcement learning as the agent can be allowed to play many thousands of games against the computer game system in order to learn a successful policy, without incurring the cost of creating and labeling a large dataset of example situations with correct joystick actions. The DeepMind system got so good at the games that it outperformed all previous computer systems on six of the seven games, and outperformed human experts on three of the games.

Deep learning can be applied to all three machine learning scenarios: supervised, unsupervised, and reinforcement. Supervised machine learning is, however, the most common type of machine learning. Consequently, the majority of this book will focus on deep learning in a supervised learning context. However, most of the deep learning concerns and principles introduced in the supervised learning context also apply to unsupervised and reinforcement learning.

Why Is Deep Learning So Successful?

In any data-driven process the primary determinant of success is knowing what to measure and how to measure it. This is why the processes of feature selection and feature design are so important to machine learning. As discussed above, these tasks can require domain expertise, statistical analysis of the data, and iterations of experiments building models with different feature sets. Consequently, dataset design and preparation can consume a significant portion of time and resources expended in the project, in some cases approaching up to 80% of the total budget of a project (Kelleher and Tierney 2018). Feature design is one task in which deep learning can have a significant advantage over traditional machine learning. In traditional machine learning, the design of features often requires a large amount of human effort. Deep learning takes a different approach to feature design, by attempting to automatically learn the features that are most useful for the task from the raw data.

In any data-driven process the primary determinant of success is knowing what to measure and how to measure it.

To give an example of feature design, a person's body mass index (BMI) is the ratio of a person's weight (in kilograms) divided by their height (in meters squared). In a medical setting, BMI is used to categorize people as underweight, normal, overweight, or obese. Categorizing people in this way can be useful in predicting the likelihood of a person developing a weight-related medical condition, such as diabetes. BMI is used for this categorization because it enables doctors to categorize people in a manner that is relevant to these weight-related medical conditions. Generally, as people get taller they also get heavier. However, most weight-related medical conditions (such as diabetes) are not affected by a person's height but rather the amount they are overweight compared to other people of a similar stature. BMI is a useful feature to use for the medical categorization of a person's weight because it takes the effect of height on weight into account. BMI is an example of a feature that is derived (or calculated) from raw features; in this case the raw features are weight and height. BMI is also an example of how a derived feature can be more useful in making a decision than the raw features that it is derived from. BMI is a hand-designed feature: Adolphe Quetelet designed it in the eighteenth century.

As mentioned above, during a machine learning project a lot of time and effort is spent on identifying, or designing, (derived) features that are useful for the task the project is trying to solve. The advantage of deep learning is that it can learn useful derived features from data automatically (we will discuss how it does this in later chapters). Indeed, given large enough datasets, deep learning has proven to be so effective in learning features that deep learning models are now more accurate than many of the other machine learning models that use hand-engineered features. This is also why deep learning is so effective in domains where examples are described with very large numbers of features. Technically datasets that contain large numbers of features are called high-dimensional. For example, a dataset of photos with a feature for each pixel in a photo would be high-dimensional. In complex high-dimensional domains, it is extremely difficult to hand-engineer features: consider the challenges of hand-engineering features for face recognition or machine translation. So, in these complex domains, adopting a strategy whereby the features are automatically learned from a large dataset makes sense. Related to this ability to automatically learn useful features, deep learning also has the ability to learn complex nonlinear mappings between inputs and outputs; we will explain the concept of a nonlinear mapping in chapter 3, and in chapter 6 we will explain how these mappings are learned from data.

Summary and the Road Ahead

This chapter has focused on positioning deep learning within the broader field of machine learning. Consequently, much of this chapter has been devoted to introducing machine learning. In particular, the concept of a function as a deterministic mapping from inputs to outputs was introduced, and the goal of machine learning was explained as finding a function that matches the mappings from input features to the output features that are observed in the examples in the dataset.

Within this machine learning context, deep learning was introduced as the subfield of machine learning that focuses on the design and evaluation of training algorithms and model architectures for modern neural networks. One of the distinctive aspects of deep learning within machine learning is the approach it takes to feature design. In most machine learning projects, feature design is a human-intensive task that can require deep domain expertise and consume a lot of time and project budget. Deep learning models, on the other hand, have the ability to learn useful features from low-level raw data, and complex nonlinear mappings from inputs to outputs. This ability is dependent on the availability of large datasets; however, when such datasets are available, deep learning can frequently outperform other machine learning approaches. Furthermore, this ability to learn useful features from large datasets is why deep learning can often generate highly accurate models for complex domains, be it in machine translation, speech processing, or image or video processing. In a sense, deep learning has unlocked the potential of big data. The most noticeable impact of this development has been the integration of deep learning models into consumer devices. However, the fact that deep learning can be used to analyze massive datasets also has implications for our individual privacy and civil liberty (Kelleher and Tierney 2018). This is why understanding what deep learning is, how it works, and what it can and can’t be used for, is so important. The road ahead is as follows:

- Chapter 2 introduces some of the foundational concepts of deep learning, including what a model is, how the parameters of a model can be set using data, and how we can create complex models by combining simple models.
- Chapter 3 explains what neural networks are, how they work, and what we mean by a deep neural network.

- Chapter 4 presents a history of deep learning. This history focuses on the major conceptual and technical breakthroughs that have contributed to the development of the field of machine learning. In particular, it provides a context and explanation for why deep learning has seen such rapid development in recent years.
- Chapter 5 describes the current state of the field, by introducing the two deep neural architectures that are the most popular today: convolutional neural networks and recurrent neural networks. Convolutional neural networks are ideally suited to processing image and video data. Recurrent neural networks are ideally suited to processing sequential data such as speech, text, or time-series data. Understanding the differences and commonalities across these two architectures will give you an awareness of how a deep neural network can be tailored to the characteristics of a specific type of data, and also an appreciation of the breadth of the design space of possible network architectures.
- Chapter 6 explains how deep neural networks models are trained, using the gradient descent and backpropagation algorithms. Understanding these two algorithms will give you a real insight into the state of artificial intelligence. For example, it will help you to understand why, given enough data, it is currently possible to train a computer to do a specific task within a well-defined domain at a level beyond human capabilities, but also why a more general form of intelligence is still an open research challenge for artificial intelligence.
- Chapter 7 looks to the future in the field of deep learning. It reviews the major trends driving the development of deep learning at present, and how they are likely to contribute to the development of the field in the coming years. The chapter also discusses some of the challenges the field faces, in particular the challenge of understanding and interpreting how a deep neural network works.

2

Conceptual Foundations

This chapter introduces some of the foundational concepts that underpin deep learning. The basis of this chapter is to decouple the initial presentation of these concepts from the technical terminology used in deep learning, which is introduced in subsequent chapters.

A deep learning network is a mathematical model that is (loosely) inspired by the structure of the brain. Consequently, in order to understand deep learning it is helpful to have an intuitive understanding of what a mathematical model is, how the parameters of a model can be set, how we can combine (or compose) models, and how we can use geometry to understand how a model processes information.

What Is a Mathematical Model?

In its simplest form, a mathematical model is an equation that describes how one or more input variables are related to an output variable. In this form a mathematical model is the same as a function: a mapping from inputs to outputs.

In any discussion relating to models, it is important to remember the statement by George Box that *all models are wrong but some are useful!* For a model to be useful it must have a correspondence with the real world. This correspondence is most obvious in terms of the meaning that can be associated with a variable. For example, in isolation a value such as 78,000 has no meaning because it has no correspondence with concepts in the real world. But *yearly income=$78,000* tells us how the number describes an aspect of the real world. Once the variables in a model have a meaning, we can understand the model as describing a process through which different aspects of the world interact and cause new events. The new events are then described by the outputs of the model.

A very simple template for a model is the equation of a line:

$$y = mx + c$$

In this equation y is the output variable, x is the input variable, and m and c are two parameters of the model that we can set to adjust the relationship the model defines between the input and the output.

Imagine we have a hypothesis that yearly income affects a person's happiness and we wish to describe the relationship between these two variables.[1] Using the equation of a line, we could define a model to describe this relationship as follows:

$$\text{happiness} = m \times \text{income} + c$$

This model has a meaning because the variables in the model (as distinct from the parameters of the model) have a correspondence with concepts from the real world. To complete our model, we have to set the values of the model's parameters: m and c. Figure 2.1 illustrates how varying the values of each of these parameters changes the relationship defined by the model between *income* and *happiness*.

One important thing to notice in this figure is that no matter what values we set the model parameters to, the relationship defined by the model between the input and the output variable can be plotted as a line. This is not surprising because we used the equation of a line as the template to define our model, and this is why mathematical models that are based on the equation of a line are known as linear models. The other important thing to notice in the figure is how changing the parameters of the model changes the relationship between income and happiness.

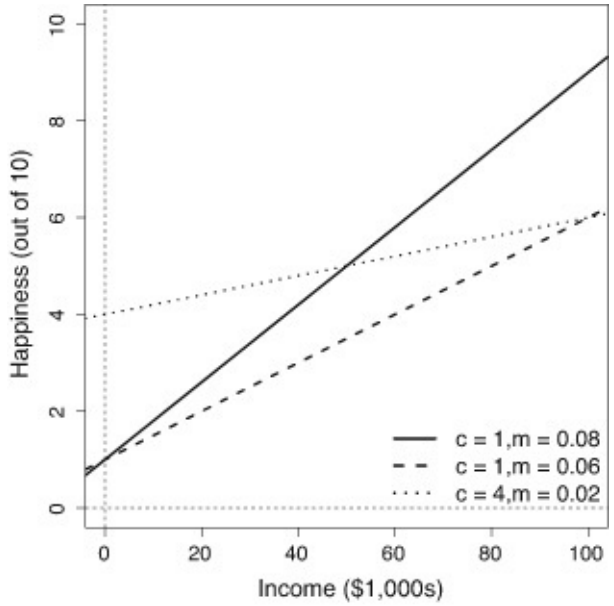


Figure 2.1 Three different linear models of how income affects happiness.

The solid steep line, with parameters $(c = 1, m = 0.08)$, is a model of the world in which people with zero income have a happiness level of 1, and increases in income have a significant effect on people's happiness. The dashed line, with parameters $(c = 1, m = 0.06)$, is a model in which people with zero income have a happiness level of 1 and increased income increases happiness, but at the slower rate compared to the world modeled by the solid line. Finally, the dotted line, parameters $(c = 4, m = 0.02)$, is a model of the world where no one is particularly unhappy—even people with zero income have a happiness of 4 out of 10—and although increases in income do affect happiness, the effect is moderate. This third model assumes that income has a relatively weak effect on happiness.

More generally, the differences between the three models in figure 2.1 show how making changes to the parameters of a linear model changes the model. Changing c causes the line to move up and done. This is most clearly seen if we

focus on the y-axis: notice that the line defined by a model always crosses (or intercepts) the y-axis at the value that c is set to. This is why the c parameter in a linear model is known as the intercept. The intercept can be understood as specifying the value of the output variable when the input variable is zero. Changing the m parameter changes the angle (or slope) of the line. The slope parameter controls how quickly changes in income effect changes in happiness. In a sense, the slope value is a measure of how important income is to happiness. If income is very important (i.e., if small changes in income result in big changes in happiness), then the slope parameter of our model should be set to a large value. Another way of understanding this is to think of a slope parameter of a linear model as describing the importance, or weight, of the input variable in determining the value of the output.

Linear Models with Multiple Inputs

The equation of a line can be used as a template for mathematical models that have more than one input variable. For example, imagine yourself in a scenario where you have been hired by a financial institution to act as a loan officer and your job involves deciding whether or not a loan application should be granted. From interviewing domain experts you come up with a hypothesis that a useful way to model a person's credit solvency is to consider both their yearly income and their current debts. If we assume that there is a linear relationship between these two input variables and a person's credit solvency, then the appropriate mathematical model, written out in English would be:

$$\begin{aligned}\text{solvency} = &(\text{income} \times \text{weight for income}) \\ &+ (\text{debt} \times \text{weight for debt}) + \text{intercept}\end{aligned}$$

Notice that in this model the m parameter has been replaced by a separate weight for each input variable, with each weight representing the importance of its associated input in determining the output. In mathematical notation this model would be written as:

$$y = (\text{input}_1 \times \text{weight}_1) + (\text{input}_2 \times \text{weight}_2) + c$$

where y represents the *credit solvency* output, input_1 represents the *income* variable, input_2 represents the *debt* variable, and c represents the intercept. Using the idea of adding a new weight for each new input to the model allows us to scale the equation of a line to as many inputs as we like. All the models defined in this way are still linear within the dimensions defined by the number of inputs and the output. What this means is that a linear model with two inputs and one output defines a flat plane rather than a line because that is what a two-dimensional line that has been extruded to three dimensions looks like.

It can become tedious to write out a mathematical model that has a lot of inputs, so mathematicians like to write things in as compact a form as possible. With this in mind, the above equation is sometimes written in the short form:

$$y = \sum_{i=1}^{n} (\text{input}_i \times \text{weight}_i) + c$$

This notation tells us that to calculate the output variable y we must first go through all n inputs and multiple each input by its corresponding weight, then we should sum together the results of these n multiplications, and finally we add the c intercept parameter to the result of the summation. The Σ symbol tells us that we use addition to combine the results of the multiplications, and the index i tells us that we multiply each input by the weight with the same index. We can make our notation even more compact by treating the intercept as a weight. One way to do this is to assume an input_0 that is always equal to 1 and to treat the intercept as the weight on this input, that is, weight_0. Doing this allows us to write out the model as follows:

$$y = \sum_{i=0}^{n} (\text{input}_i \times \text{weight}_i)$$

Notice that the index now starts at 0, rather than 1, because we are now assuming an extra input, $\text{input}_0 = 1$, and we have relabeled the intercept weight_0.

Although we can write down a linear model in a number of different ways, the core of a linear model is that the output is calculated as the *sum of the n input values multiplied by their corresponding weights*. Consequently, this type of model defines a calculation known as a *weighted sum*, because we weight each input and sum the results. Although a weighted sum is easy to calculate, it turns out to be very useful in many situations, and it is the basic calculation used in every neuron in a neural network.

Setting the Parameters of a Linear Model

Let us return to our working scenario where we wish to create a model that enables us to calculate the credit solvency of individuals who have applied for a financial loan. For simplicity in presentation we will ignore the intercept parameter in this discussion as it is treated the same as the other parameters (i.e., the weights on the inputs). So, dropping the intercept parameter, we have the following linear model (or weighted sum) of the relationship between a person's income and debt to their credit solvency:

The multiplication of inputs by weights, followed by a summation, is known as a *weighted sum.*

$$\begin{aligned}\text{solvency} = &(\text{income} \times \text{weight for income}) \\ &+ (\text{debt} \times \text{weight for debt})\end{aligned}$$

In order to complete our model, we need to specify the parameters of the model; that is, we need to specify the value of the weight for each input. One way to do this would be to use our domain expertise to come up with values for each of the parameters.

For example, if we assume that an increase in a person's income has a bigger impact on their credit solvency than a similar increase in their debt, we should set the weighting for income to be larger than that of the debt. The following model encodes this assumption; in particular this model specifies that income is three times as important as debt in determining a person's credit solvency:

$$\text{solvency} = (\text{income} \times 3) + (\text{debt} \times 1)$$

The drawback with using domain knowledge to set the parameters of a model is that experts often disagree. For example, you may think that weighting income as three times as important as debt is not realistic; in that case the model can be adjusted by, for example, setting both income and debt to have an equal weighting, which would be equivalent to assuming that income and debt are equally important in determining credit solvency. One way to avoid arguments between experts is to use data to set the parameters. This is where machine

learning helps. The learning done by machine learning is finding the parameters (or weights) of a model using a dataset.

Learning Model Parameters from Data

Later in the book we will describe the standard algorithm used to learn the weights for a linear model, known as the gradient descent algorithm. However, we can give a brief preview of the algorithm here. We start with a dataset containing a set of examples for which we have both the input values (income and debt) and the output value (credit solvency). Table 2.1 illustrates such a dataset from our credit solvency scenario.[2]

The learning done by machine learning is finding the parameters (or weights) of a model using a dataset.

We then begin the process of learning the weights by guessing initial values for each weight. It is very likely that this initial, guessed, model will be a very bad model. This is not a problem, however, because we will use the dataset to iteratively update the weights so that the model gets better and better, in terms of how well it matches the data. For the purpose of the example, we will use the model described above as our initial (guessed) model:

Table 2.1. A dataset of loan applications and known credit solvency rating of the applicant

ID	Annual income	Current debt	Credit solvency
1	$150	-$100	100
2	$250	-$300	-50
3	$450	-$250	400
4	$200	-$350	-300

$$\text{solvency} = (\text{income} \times 3) + (\text{debt} \times 1)$$

The general process for improving the weights of the model is to select an example from the dataset and feed the input values from the example into the model. This allows us to calculate an estimate of the output value for the example. Once we have this estimated output, we can calculate the error of the model on the example by subtracting the estimated output from the correct output for the example listed in the dataset. Using the error of the model on the example, we can improve how well the model fits the data by updating the weights in the model using the following strategy, or learning rule:

- If the error is 0, then we should not change the weights of the model.
- If the error is positive, then the output of the model was too low, so we should increase the output of the model for this example by increasing the weights for all the inputs that had positive values for the example and decreasing the weights for all the inputs that had negative values for the example.
- If the error is negative, then the output of the model was too high, so we should decrease the output of the model for this example by decreasing the weights for all the inputs that had positive values for the example and increasing the weights for all the inputs that had negative values for the example.

To illustrate the weight update process we will use example 1 from table 2.1 (income = 150, debt = -100, and solvency = 100) to test the accuracy of our guessed model and update the weights according to the resulting error.

$$\begin{aligned}\text{solvency} &= (\text{income} \times 3) + (\text{debt} \times 1) \\ &= (150 \times 3) + (-100 \times 1) \\ &= 350\end{aligned}$$

When the input values for the example are passed into the model, the credit solvency estimate returned by the model is 350. This is larger than the credit solvency listed for this example in the dataset, which is 100. As a result, the error of the model is negative (100 – 350 = –250); therefore, following the learning rule described above, we should decrease the output of the model for this example by decreasing the weights for positive inputs and increasing the weights for negative inputs. For this example, the income input had a positive value and the debt input had a negative value. If we decrease the weight for income by 1 and increase the weight for debt by 1, we end up with the following

model:

$$\text{solvency} = (\text{income} \times 2) + (\text{debt} \times 2)$$

We can test if this weight update has improved the model by checking if the new model generates a better estimate for the example than the old model. The following illustrates pushing the same example through the new model:

$$\begin{aligned}\text{solvency} &= (\text{income} \times 2) + (\text{debt} \times 2) \\ &= (150 \times 2) + (-100 \times 2) \\ &= 100\end{aligned}$$

This time the credit solvency estimate generated by the model matches the value in the dataset, showing that the updated model fits the data more closely than the original model. In fact, this new model generates the correct output for all the examples in the dataset.

In this example, we only needed to update the weights once in order to find a set of weights that made the behavior of the model consistent with all the examples in the dataset. Typically, however, it takes many iterations of presenting examples and updating weights to get a good model. Also, in this example, we have, for the sake of simplicity, assumed that the weights are updated by either adding or subtracting 1 from them. Generally, in machine learning, the calculation of how much to update each weight by is more complicated than this. However, these differences aside, the general process outlined here for updating the weights (or parameters) of a model in order to fit the model to a dataset is the learning process at the core of deep learning.

Combining Models

We now understand how we can specify a linear model to estimate an applicant's credit solvency, and how we can modify the parameters of the model in order to fit the model to a dataset. However, as a loan officer our job is not simply to calculate an applicant's credit solvency; we have to decide whether to grant the loan application or not. In other words, we need a rule that will take a credit solvency score as input and return a decision on the loan application. For example, we might use the decision rule that *a person with a credit solvency above 200 will be granted a loan*. This decision rule is also a model: it maps an input variable, in this case *credit solvency*, to an output variable, *loan decision*.

Using this decision rule we can adjudicate on a loan application by first using the model of credit solvency to convert a loan applicant's profile (described in terms of the annual income and debt) into a credit solvency score, and then passing the resulting credit solvency score through our decision rule model to generate the loan decision. We can write this process out in a pseudomathematical shorthand as follows:

$$\begin{aligned}&\text{loan decision}\\&\quad=\text{decision rule}(\text{solvency}=(\text{income}\times 2)+(\text{debt}\times 2))\end{aligned}$$

Using this notation, the entire decision process for adjudicating the loan application for example 1 from table 2.1 is:

$$\begin{aligned}&\text{loan decision}\\&\quad=\text{decision rule}(\text{solvency}=(\text{income}\times 2)+(\text{debt}\times 2))\\&=\text{decision rule}(\text{solvency}=(150\times 2)+(-100\times 2))\\&=\text{decision rule}(\text{solvency}=100)\\&=\text{reject}\end{aligned}$$

We are now in a position where we can use a model (composed of two simpler models, a decision rule and a weighted sum) to describe how a loan decision is made. What is more, if we use data from previous loan applications to set the parameters (i.e., the weights) of the model, our model will correspond to how we have processed previous loan applications. This is useful because we can use this

model to process new applications in a way that is consistent with previous decisions. If a new loan application is submitted, we simply use our model to process the application and generate a decision. It is this ability to apply a mathematical model to new examples that makes mathematical modeling so useful.

When we use the output of one model as the input to another model, we are creating a third model by combining two models. This strategy of building a complex model by combining smaller simpler models is at the core of deep learning networks. As we will see, a neural network is composed of a large number of small units called neurons. Each of these neurons is a simple model in its own right that maps from a set of inputs to an output. The overall model implemented by the network is created by feeding the outputs from one group of neurons as inputs into a second group of neurons and then feeding the outputs of the second group of neurons as inputs to a third group of neurons, as so on, until the final output of the model is generated. The core idea is that feeding the outputs of some neuron as inputs to other neurons enables these subsequent neurons to learn to solve a different part of the overall problem the network is trying to solve by building on the partial solutions implemented by the earlier neurons—in a similar way to the way the decision rule generates the final adjudication for a loan application by building on the calculation of the credit solvency model. We will return to this topic of model composition in subsequent chapters.

Input Spaces, Weight Spaces, and Activation Spaces

Although mathematical models can be written out as equations, it is often useful to understand the geometric meaning of a model. For example, the plots in figure 2.1 helped us understand how changes in the parameters of a linear model changed the relationship between the variables that the model defined. There are a number of geometric spaces that it is useful to distinguish between, and understand, when we are discussing neural networks. These are the input space, the weight space, and the activation space of a neuron. We can use the decision model for loan applications that we defined in the previous section to explain these three different types of spaces.

We will begin by describing the concept of an input space. Our loan decision model took two inputs: the annual income and current debt of the applicant. Table 2.1 listed these input values for four example loan applications. We can plot the input space of this model by treating each of the input variables as the axis of a coordinate system. This coordinate space is referred to as the input space because each point in this space defines a possible combination of input values to the model. For example, the plot at the top-left of figure 2.2 shows the position of each of the four example loan applications within the models input space.

The weight space for a model describes the universe of possible weight combinations that a model might use. We can plot the weight space for a model by defining a coordinate system with one axis per weight in the model. The loan decision model has only two weights, one weight for the annual income input, and one weight for the current debt input. Consequently, the weight space for this model has two dimensions. The plot at the top-right of figure 2.2 illustrates a portion of the weight space for this model. The location of the weight combination used by the model ^{2,2} is highlighted in this figure. Each point within this coordinate system describes a possible set of weights for the model, and therefore corresponds to a different weighted sum function within the model. Consequently, moving from one location to another within this weight space is equivalent to changing the model because it changes the mapping from inputs to output that the model defines.

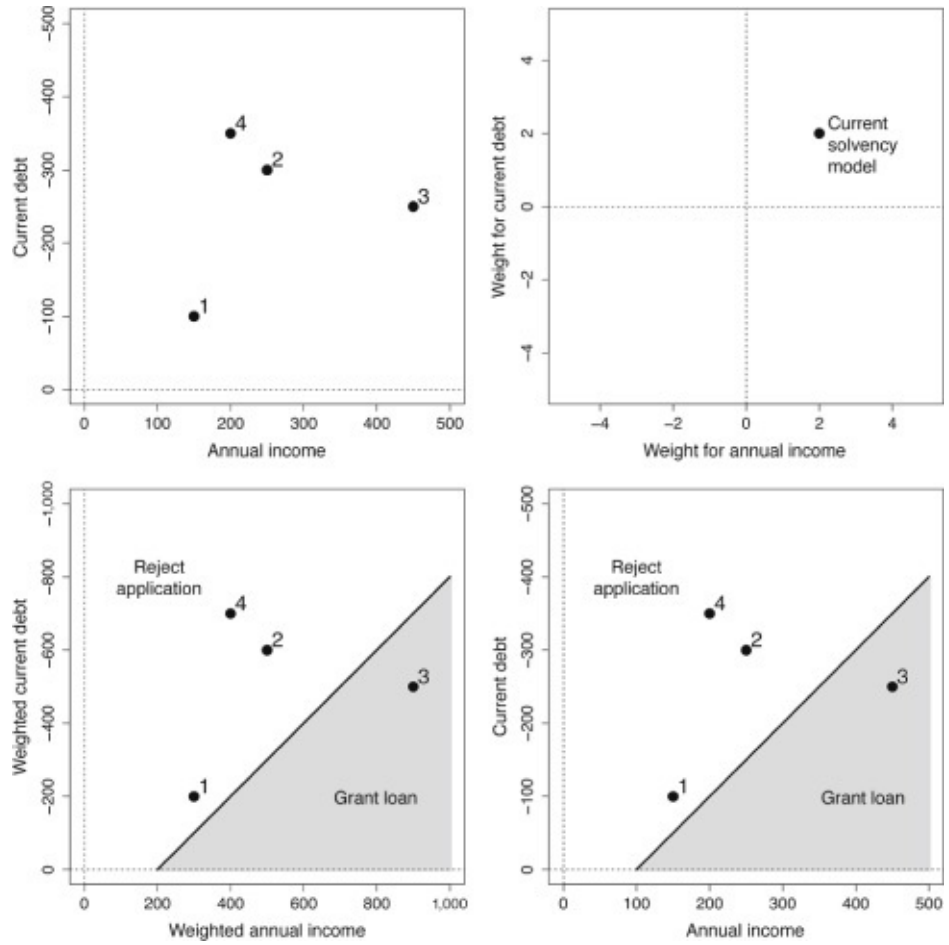


Figure 2.2 There are four different coordinate spaces related to the processing of the loan decision model: top-left plots the input space; top-right plots the weight space; bottom-left plots the activation (or decision) space; and bottom-right plots the input space with the decision boundary plotted.

A linear model maps a set of input values to a point in a new space by applying a weighted sum calculation to the inputs: multiply each input by a weight, and sum the results of the multiplication. In our loan decision model it is in this space that we apply our decision rule. Thus, we could call this space the decision space, but, for reasons that will become clear when we describe the structure of a neuron in the next chapter, we call this space the activation space. The axes of a model's activation space correspond to the weighted inputs to the model. Consequently, each point in the activation space defines a set of weighted inputs. Applying a decision rule, such as our rule that *a person with a credit solvency above 200 will be granted a loan*, to each point in this activation space, and recording the result of the decision for each point, enables us to plot the decision boundary of the model in this space. The decision boundary divides those points in the activation space that exceed the threshold, from those points in the space below the threshold. The plot in the bottom-left of figure 2.2

illustrates the activation space for our loan decision model. The positions of the four example loan applications listed in table 2.1 when they are projected into this activation space are shown. The diagonal black line in this figure shows the decision boundary. Using this threshold, loan application number three is granted and the other loan applications are rejected. We can, if we wish, project the decision boundary back into the original input space by recording for each location in the input space which side of the decision boundary in the activation space it is mapped to by the weighted sum function. The plot at the bottom-right of figure 2.2 shows the decision boundary in the original input space (note the change in the values on the axes) and was generated using this process. We will return to the concepts of weight spaces and decision boundaries in next chapter when we describe how adjusting the parameters of a neuron changes the set of input combinations that cause the neuron to output a high activation.

Summary

The main idea presented in this chapter is that a linear mathematical model, be it expressed as an equation or plotted as a line, describes a relationship between a set of inputs and an output. Be aware that not all mathematical models are linear models, and we will come across nonlinear models in this book. However, the fundamental calculation of a weighted sum of inputs does define a linear model. Another big idea introduced in this chapter is that a linear model (a weighted sum) has a set of parameters, that is, the weights used in the weighted sum. By changing these parameters we can change the relationship the model describes between the inputs and the output. If we wish we could set these weights by hand using our domain expertise; however, we can also use machine learning to set the weights of the model so that the behavior of the model fits the patterns found in a dataset. The last big idea introduced in this chapter was that we can build complex models by combining simpler models. This is done by using the output from one (or more) models as input(s) to another model. We used this technique to define our composite model to make loan decisions. As we will see in the next chapter, the structure of a neuron in a neural network is very similar to the structure of this loan decision model. Just like this model, a neuron calculates a weighted sum of its inputs and then feeds the result of this calculation into a second model that decides whether the neuron activates or not.

The focus of this chapter has been to introduce some foundational concepts before we introduce the terminology of machine learning and deep learning. To give a quick overview of how the concepts introduced in this chapter map over to machine learning terminology, our loan decision model is equivalent to a two-input neuron that uses a threshold activation function. The two financial indicators (annual income and current debt) are analogous to the inputs the neuron receives. The terms input vector or feature vector are sometimes used to refer to the set of indicators describing a single example; in this context an example is a single loan applicant, described in terms of two features: annual income and current debt. Also, just like the loan decision model, a neuron associates a weight with each input. And, again, just like the loan decision model, a neuron multiplies each input by its associated weight and sums the results of these multiplications in order to calculate an overall score for the inputs. Finally, similar to the way we applied a threshold to the credit solvency score to convert it into a decision of whether to grant or reject the loan application, a neuron applies a function (known as an activation function) to

convert the overall score of the inputs. In the earliest types of neurons, these activation functions were actually threshold functions that worked in exactly the same way as the score threshold used in this credit scoring example. In more recent neural networks, different types of activation functions (for example, the logistic, tanh, or ReLU functions) are used. We will introduce these activation functions in the next chapter.

3

Neural Networks: The Building Blocks of Deep Learning

The term *deep learning* describes a family of *neural network* models that have multiple layers of simple information processing programs, known as neurons, in the network. The focus of this chapter is to provide a clear and comprehensive introduction to how these neurons work and are interconnected in artificial neural networks. In later chapters, we will explain how neural networks are trained using data.

A neural network is a computational model that is inspired by the structure of the human brain. The human brain is composed of a massive number of nerve cells, called neurons. In fact, some estimates put the number of neurons in the human brain at one hundred billion (Herculano-Houzel 2009). Neurons have a simple three-part structure consisting of: a cell body, a set of fibers called dendrites, and a single long fiber called an axon. Figure 3.1 illustrates the structure of a neuron and how it connects to other neurons in the brain. The dendrites and the axon stem from the cell body, and the dendrites of one neuron are connected to the axons of other neurons. The dendrites act as input channels to the neuron and receive signals sent from other neurons along their axons. The axon acts as the output channel of a neuron, and so other neurons, whose dendrites are connected to the axon, receive the signals sent along the axon as inputs.

Neurons work in a very simple manner. If the incoming stimuli are strong enough, the neuron transmits an electrical pulse, called an action potential, along its axon to the other neurons that are connected to it. So, a neuron acts as an all-or-none switch, that takes in a set of inputs and either outputs an action potential or no output.

This explanation of the human brain is a significant simplification of the biological reality, but it does capture the main points necessary to understand the analogy between the structure of the human brain and computational models called neural networks. These points of analogy are: (1) the brain is composed of a large number of interconnected and simple units called neurons; (2) the functioning of the brain can be understood as processing information, encoded as high or low electrical signals, or activation potentials, that spread across the

network of neurons; and (3) each neuron receives a set of stimuli from its neighbors and maps these inputs to either a high- or low-value output. All computational models of neural networks have these characteristics.

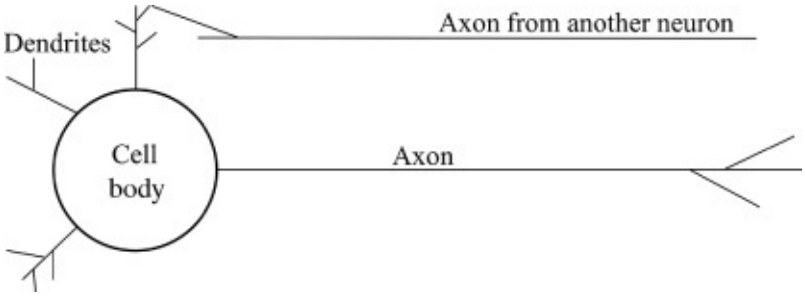


Figure 3.1 The structure of a neuron in the brain.

Artificial Neural Networks

An artificial neural network consists of a network of simple information processing units, called neurons. The power of neural networks to model complex relationships is not the result of complex mathematical models, but rather emerges from the interactions between a large set of simple neurons.

Figure 3.2 illustrates the structure of a neural network. It is standard to think of the neurons in a neural network as organized into layers. The depicted network has five layers: one input layer, three hidden layers, and one output layer. A hidden layer is just a layer that is neither the input nor the output layer. Deep learning networks are neural networks that have many hidden layers of neurons. The minimum number of hidden layers necessary to be considered deep is two. However, most deep learning networks have many more than two hidden layers. The important point is that the depth of a network is measured in terms of the number of hidden layers, plus the output layer.

Deep learning networks are neural networks that have many hidden layers of neurons.

In figure 3.2, the squares in the input layer represent locations in memory that are used to present inputs to the network. These locations can be thought of as sensing neurons. There is no processing of information in these sensing neurons; the output of each of these neurons is simply the value of the data stored at the memory location. The circles in the figure represent the information processing neurons in the network. Each of these neurons takes a set of numeric values as input and maps them to a single output value. Each input to a processing neuron is either the output of a sensing neuron or the output of another processing neuron.

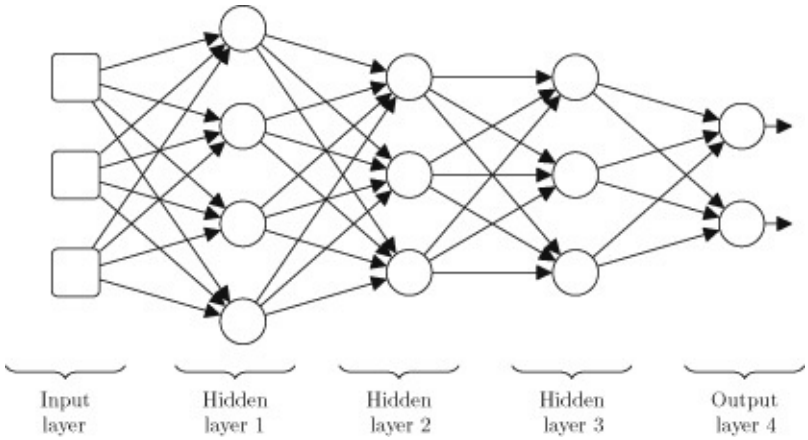


Figure 3.2 Topological illustration of a simple neural network.

The arrows in figure 3.2 illustrate how information flows through the network from the output of one neuron to the input of another neuron. Each connection in a network connects two neurons and each connection is directed, which means that information carried along a connection only flows in one direction. Each of the connections in a network has a *weight* associated with it. A connection weight is simply a number, but these weights are very important. The weight of a connection affects how a neuron processes the information it receives along the connection, and, in fact, training an artificial neural network, essentially, involves searching for the best (or optimal) set of weights.

How an Artificial Neuron Processes Information

The processing of information within a neuron, that is, the mapping from inputs to an output, is very similar to the loan decision model that we developed in chapter 2. Recall that the loan decision model first calculated a weighted sum over the input features (income and debt). The weights used in the weighted sum were adjusted using a dataset so that the results of the weighted sum calculation, given an loan applicant's income and debt as inputs, was an accurate estimate of the applicant's credit solvency score. The second stage of processing in the loan decision model involved passing the result of the weighted sum calculation (the estimated credit solvency score) through a decision rule. This decision rule was a function that mapped a credit solvency score to a decision on whether a loan application was granted or rejected.

A neuron also implements a two-stage process to map inputs to an output. The first stage of processing involves the calculation of a weighted sum of the inputs to the neuron. Then the result of the weighted sum calculation is passed through a second function that maps the results of the weighted sum score to the neuron's final output value. When we are designing a neuron, we can used many different types of functions for this second stage or processing; it may be as simple as the decision rule we used for our loan decision model, or it may be more complex. Typically the output value of a neuron is known as its activation value, so this second function, which maps from the result of the weighted sum to the activation value of the neuron, is known as an activation function.

Figure 3.3 illustrates how these stages of processing are reflected in the structure of an artificial neuron. In figure 3.3, the Σ symbol represents the calculation of the weighted sum, and the φ symbol represents the activation function processing the weighted sum and generating the output from the neuron.

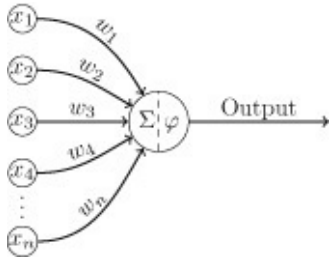


Figure 3.3 The structure of an artificial neuron.

The neuron in figure 3.3 receives *n* inputs $[x_1, \ldots, x_n]$ on *n* different input connections, and each connection has an associated weight $[w_1, \ldots, w_n]$. The weighted sum calculation involves the multiplication of inputs by weights and the summation of the resulting values. Mathematically this calculation is written as:

$$z = (x_1 \times w_1) + (x_2 \times w_2) + \ldots + (x_n \times w_n)$$

This calculation can also be written in a more compact mathematical form as:

$$z = \sum_{i=1}^{n} x_i \times w_i$$

For example, assuming a neuron received the inputs $[x_1 = 3, x_2 = 9]$ and had the following weights $[w_1 = -3, \ w_2 = 1]$, the weighted sum calculation would be:

$$z = (3 \times -3) + (9 \times 1)$$
$$= 0$$

The second stage of processing within a neuron is to pass the result of the weighted sum, the z value, through an *activation function*. Figure 3.4 plots the shape of a number of possible activation functions, as the input to each function, z, ranges across an interval, either [-1, ..., +1] or [-10, ..., +10] depending on which interval best illustrates the shape of the function. Figure 3.4 (top) plots a threshold activation function. The decision rule we used in the loan decision model was an example of a threshold function; the threshold used in that decision rule was whether the credit solvency score was above 200. Threshold activations were common in early neural network research. Figure 3.4 (middle) plots the *logistic* and *tanh* activation functions. The units employing these activation functions were popular in multilayer networks until quite recently. Figure 3.4 (bottom) plots the rectifier (or hinge, or positive linear) activation function. This activation function is very popular in modern deep learning networks; in 2011 the rectifier activation function was shown to enable better training in deep networks (Glorot et al. 2011). In fact, as will be discussed in chapter 4, during the review of the history of deep learning, one of the trends in neural network research has been a shift from threshold activation to logistic and tanh activations, and then onto rectifier activation functions.

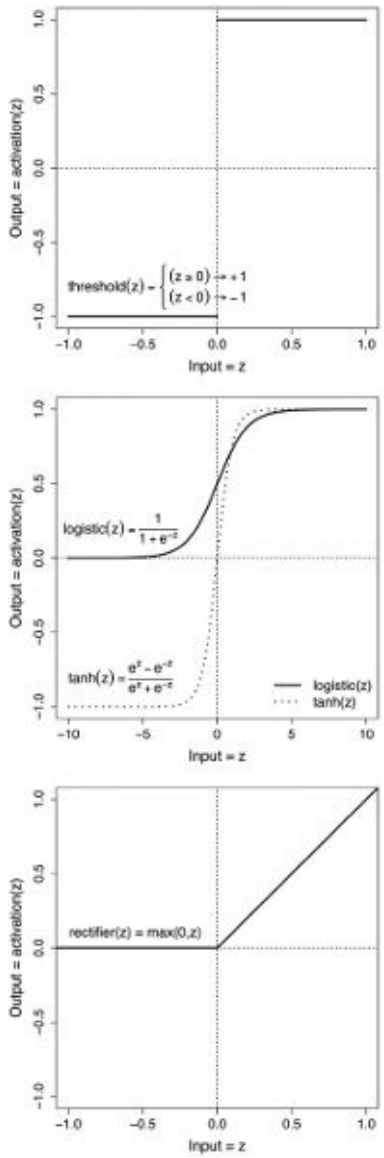


Figure 3.4 Top: threshold function; middle: logistic and tanh functions; bottom: rectified linear function.

Returning to the example, the result of the weighted summation step was $z = 0$. Figure 3.4 (middle plot, solid line) plots the logistic function. Assuming that the neuron is using a logistic activation function, this plot shows how the result of the summation will be mapped to an output activation: $\text{logistic}(0) = 0.5$. The calculation of the output activation of this neuron can be summarized as:

$$\text{Output} = \text{activation\_function}\left(z = \sum_{i=1}^{n} x_i \times w_i\right)$$

$$= \text{logistic}(z = (3 \times -3) + (9 \times 1))$$

$$= \text{logistic}(z = 0)$$
$$= 0.5$$

Notice that the processing of information in this neuron is nearly identical to the processing of information in the loan decision model we developed in the last chapter. The major difference is that we have replaced the decision threshold rule that mapped the weighted sum score to an accepted or rejected output with a logistic function that maps the weighted sum score to a value between 0 and 1. Depending on the location of this neuron in the network, the output activation of the neuron, in this instance $y = 0.5$, will either be passed as input to one or more neurons in the next layer in the network, or will be part of the overall output of the network. If a neuron is at the output layer, the interpretation of what its output value means would be dependent on the task that the neuron is designed to model. If a neuron is in one of the hidden layers of the network, then it may not be possible to put a meaningful interpretation on the output of the neuron apart from the general interpretation that it represents some sort of derived feature (similar to the BMI feature we discussed in chapter 1) that the network has found useful in generating its outputs. We will return to the challenge of interpreting the meaning of activations within a neural network in chapter 7.

The key point to remember from this section is that a neuron, the fundamental building block of neural networks and deep learning, is defined by a simple two-step sequence of operations: calculating a weighted sum and then passing the result through an activation function.

Figure 3.4 illustrates that neither the tanh nor the logistic function is a linear function. In fact, the plots of both of these functions have a distinctive s-shaped (rather than linear) profile. Not all activation functions have an s-shape (for example, the threshold and rectifier are not s-shaped), but all activation functions do apply a nonlinear mapping to the output of the weighted sum. In fact, it is the introduction of the nonlinear mapping into the processing of a neuron that is the reason why activation functions are used.

Why Is an Activation Function Necessary?

To understand why a nonlinear mapping is needed in a neuron, it is first necessary to understand that, essentially, all a neural network does is define a mapping from inputs to outputs, be it from a game position in Go to an evaluation of that position, or from an X-ray to a diagnosis of a patient. Neurons are the basic building blocks of neural networks, and therefore they are the basic building blocks of the mapping a network defines. The overall mapping from inputs to outputs that a network defines is composed of the mappings from inputs to outputs that each of the neurons within the network implement. The implication of this is that if all the neurons within a network were restricted to linear mappings (i.e., weighted sum calculations), the overall network would be restricted to a linear mapping from inputs to outputs. However, many of the relationships in the world that we might want to model are nonlinear, and if we attempt to model these relationships using a linear model, then the model will be very inaccurate. Attempting to model a nonlinear relationship with a linear model would be an example of the underfitting problem we discussed in chapter 1: underfitting occurs when the model used to encode the patterns in a dataset is too simple and as a result it is not accurate.

A linear relationship exists between two things when an increase in one always results in an increase or decrease in the other at a constant rate. For example, if an employee is on a fixed hourly rate, which does not vary at weekends or if they do overtime, then there is a linear relationship between the number of hours they work and their pay. A plot of their hours worked versus their pay will result in a straight line; the steeper the line the higher their fixed hourly rate of pay. However, if we make the payment system for our hypothetical employee just slightly more complex, by, for example, increasing their hourly rate of pay when they do overtime or work weekends, then the relationship between the number of hours they work and their pay is no longer linear. Neural networks, and in particular deep learning networks, are typically used to model relationships that are much more complex than this employee's pay. Modeling these relationships accurately requires that a network be able to learn and represent complex nonlinear mappings. So, in order to enable a neural network to implement such nonlinear mappings, a nonlinear step (the activation function) must be included within the processing of the neurons in the network.

In principle, using any nonlinear function as an activation function enables a neural network to learn a nonlinear mapping from inputs to outputs. However, as

we shall see later, most of the activation functions plotted in figure 3.4 have nice mathematical properties that are helpful when training a neural network, and this is why they are so popular in neural network research.

The fact that the introduction of a nonlinearity into the processing of the neurons enables the network to learn a nonlinear mapping between input(s) and output is another illustration of the fact that the overall behavior of the network emerges from the interactions of the processing carried out by individual neurons within the network. Neural networks solve problems using a divide-and-conquer strategy: each of the neurons in a network solves one component of the larger problem, and the overall problem is solved by combining these component solutions. An important aspect of the power of neural networks is that during training, as the weights on the connections within the network are set, the network is in effect learning a decomposition of the larger problem, and the individual neurons are learning how to solve and combine solutions to the components within this problem decomposition.

Within a neural network, some neurons may use different activation functions from other neurons in the network. Generally, however, all the neurons within a given layer of a network will be of the same type (i.e., they will all use the same activation function). Also, sometimes neurons are referred to as units, with a distinction made between units based on the activation function the units use: neurons that use a threshold activation function are known as threshold units, units that use a logistic activation function are known as logistic units, and neurons that use the rectifier activation function are known as rectified linear units, or ReLUs. For example, a network may have a layer of ReLUs connected to a layer of logistic units. The decision regarding which activation functions to use in the neurons in a network is made by the data scientist who is designing the network. To make this decision, a data scientist may run a number of experiments to test which activation functions give the best performance on a dataset. However, frequently data scientists default to using whichever activation function is popular at a given point. For example, currently ReLUs are the most popular type of unit in neural networks, but this may change as new activation functions are developed and tested. As we will discuss at the end of this chapter, the elements of a neural network that are set manually by the data scientist prior to the training process are known as hyperparameters.

Neural networks solve problems using a divide-and-conquer strategy: each of the neurons in a network solves one component of the larger problem, and the overall problem is solved by combining these component solutions.

The term hyperparameter is used to describe the manually fixed parts of the model in order to distinguish them from the parameters of the model, which are the parts of the model that are set automatically, by the machine learning algorithm, during the training process. The parameters of a neural network are the weights used in the weighted sum calculations of the neurons in the network. As we touched on in chapters 1 and 2, the standard training process for setting the parameters of a neural network is to begin by initializing the parameters (the network's weights) to random values, and during training to use the performance of the network on the dataset to slowly adjust these weights so as to improve the accuracy of the model on the data. Chapter 6 describes the two algorithms that are most commonly used to train a neural network: the gradient descent algorithm and the backpropagation algorithm. What we will focus on next is understanding how changing the parameters of a neuron affects how the neuron responds to the inputs it receives.

How Does Changing the Parameters of a Neuron Affect Its Behavior?

The parameters of a neuron are the weights the neuron uses in the weighted sum calculation. Although the weighted sum calculation in a neuron is the same weighted sum used in a linear model, in a neuron the relationship between the weights and the final output of neuron is more complex because the result of the weighted sum is passed through an activation function in order to generate the final output. To understand how a neuron makes a decision on a given input, we need to understand the relationship between the neuron's weights, the input it receives, and the output it generates in response.

The relationship between a neuron's weights and the output it generates for a given input is most easily understood in neurons that use a threshold activation function. A neuron using this type of activation function is equivalent to our loan decision model that used a decision rule to classify the credit solvency scores, generated by the weighted sum calculation, to reject or grant loan applications. At the end of chapter 2, we introduced the concepts of an input space, a weight space, and an activation space (see figure 2.2). The input space for our two-input loan decision model could be visualized as a two-dimensional space, with one input (annual income) plotted along the x-axis, and the other input (current debt) on the y-axis. Each point in this plot defined a potential combination of inputs to the model, and the set of points in the input space defines the set of possible inputs the model could process. The weights used in the loan decision model can be understood as dividing the input space into two regions: the first region contains all of the inputs that result in the loan application being granted, and the other region contains all the inputs that result in the loan application being rejected. In that scenario, changing the weights used by the decision model would change the set of loan applications that were accepted or rejected. Intuitively, this makes sense because it changes the weighting that we put on an applicant's income relative to their debt when we are deciding on granting the loan or not.

We can generalize the above analysis of the loan decision model to a neuron in a neural network. The equivalent neuron structure to the loan decision model is a two-input neuron with a threshold activation function. The input space for such a neuron has a similar structure to the input space for a loan decision model. Figure 3.5 presents three plots of the input space for a two-input neuron

using a threshold function that outputs a high activation if the weighted sum result is greater than zero, and a low activation otherwise. The differences between each of the plots in this figure is that the neuron defines a different decision boundary in each case. In each plot, the decision boundary is marked with a black line.

Each of the plots in figure 3.5 was created by first fixing the weights of the neuron and then for each point in the input space recording whether the neuron returned a high or low activation when the coordinates of the point were used as the inputs to the neuron. The input points for which the neuron returned a high activation are plotted in gray, and the other points are plotted in white. The only difference between the neurons used to create these plots was the weights used in calculating the weighted sum of the inputs. The arrow in each plot illustrates the weight vector used by the neuron to generate the plot. In this context, a vector describes the direction and distance of a point from the origin.[1] As we shall see, interpreting the set of weights used by a neuron as defining a vector (an arrow from the origin to the coordinates of the weights) in the neuron's input space is useful in understanding how changes in the weights change the decision boundary of the neuron.

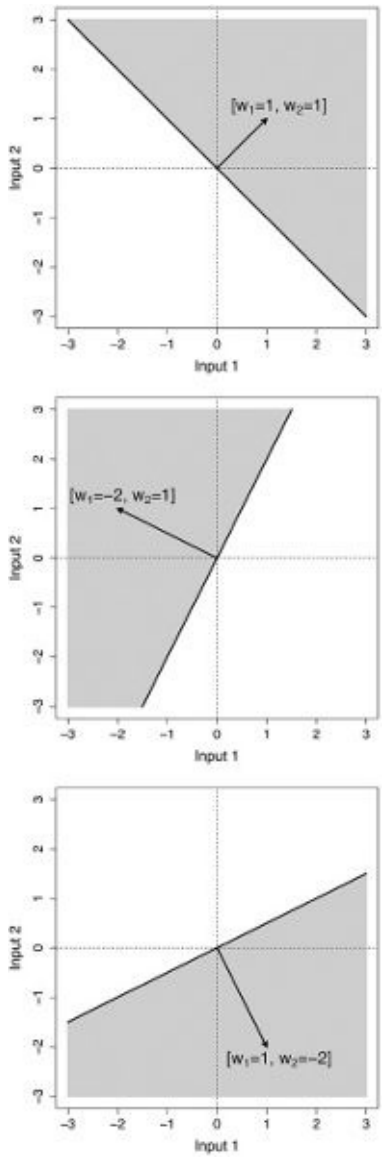


Figure 3.5 Decision boundaries for a two-input neuron. Top: weight vector [w1=1, w2=1]; middle: weight vector [w1=-2, w2=1]; bottom: weight vector [w1=1, w2=-2].

The weights used to create each plot change from one plot to the next. These changes are reflected in the direction of the arrow (the weight vector) in each plot. Specifically, changing the weights rotates the weight vector around the origin. Notice that the decision boundary in each plot is sensitive to the direction of the weight vector: in all the plots, the decision boundary is orthogonal (i.e., at a right, or 90°, angle) to the weight vector. So, changing the weights not only rotates the weight vector, it also rotates the decision boundary of the neuron. This rotation changes the set of inputs that the neuron outputs a high activation in response to (the gray regions).

To understand why this decision boundary is always orthogonal to the weight vector, we have to shift our perspective, for a moment, to linear algebra. Remember that every point in the input space defines a potential combination of input values to the neuron. Now, imagine each of these sets of input values as defining an arrow from the origin to the coordinates of the point in the input space. There is one arrow for each point in the input space. Each of these arrows is very similar to the weight vector, except that it points to the coordinates of the inputs rather than to the coordinates of the weights. When we treat a set of inputs as a vector, the weighted sum calculation is the same as multiplying two vectors, the input vector by the weight vector. In linear algebra terminology, multiplying two vectors is known as the dot product operation. For the purposes of this discussion, all we need to know about the dot product is that the result of this operation is dependent on the angle between the two vectors that are multiplied. If the angle between the two vectors is less than a right angle, then the result will be positive; otherwise, it will be negative. So, multiplying the weight vector by an input vector will return a positive value for all the input vectors at an angle less than a right angle to the weight vector, and a negative value for all the other vectors. The activation function used by this neuron returns a high activation when positive values are input and a low activation when negative values are input. Consequently, the decision boundary lies at a right angle to the weight vector because all the inputs at an angle less than a right angle to the weight vector will result in a positive input to the activation function and, therefore, trigger a high-output activation from the neuron; conversely, all the other inputs will result in a low-output activation from the neuron.

Switching back to the plots in figure 3.5, although the decision boundaries in each of the plots are at different angles, all the decision boundaries go through the point in space that the weight vectors originate from (i.e., the origin). This illustrates that changing the weights of a neuron rotates the neuron’s decision boundary but does not translate it. Translating the decision boundary means moving the decision boundary up and down the weight vector, so that the point where it meets the vector is not the origin. The restriction that all decision boundaries must pass through the origin limits the distinctions that a neuron can learn between input patterns. The standard way to overcome this limitation is to extend the weighted sum calculation so that it includes an extra element, known as the *bias term*. This bias term is not the same as the inductive bias we discussed in chapter 1. It is more analogous to the intercept parameter in the equation of a line, which moves the line up and down the y-axis. The purpose of this bias term is to move (or translate) the decision boundary away from the

origin.

The bias term is simply an extra value that is included in the calculation of the weighted sum. It is introduced into the neuron by adding the bias to the result of the weighted summation prior to passing it through the activation function. Here is the equation describing the processing stages in a neuron with the bias term represented by the term b:

$$\text{Output} = \text{activation\_function}\left(z = \underbrace{\left(\sum_{i=1}^{n} x_i \times w_i \right)}_{\text{weighted sum}} + \underbrace{b}_{\text{bias}} \right)$$

Figure 3.6 illustrates how the value of the bias term affects the decision boundary of a neuron. When the bias term is negative, the decision boundary is moved away from the origin in the direction that the weight vector points to (as in the top and middle plots in figure 3.6); when the bias term is positive, the decision boundary is translated in the opposite direction (see the bottom plot of figure 3.6). In both cases, the decision boundary remains orthogonal to the weight vector. Also, the size of the bias term affects the amount the decision boundary is moved from the origin; the larger the value of the bias term, the more the decision boundary is moved (compare the top plot of figure 3.6 with the middle and bottom plots).

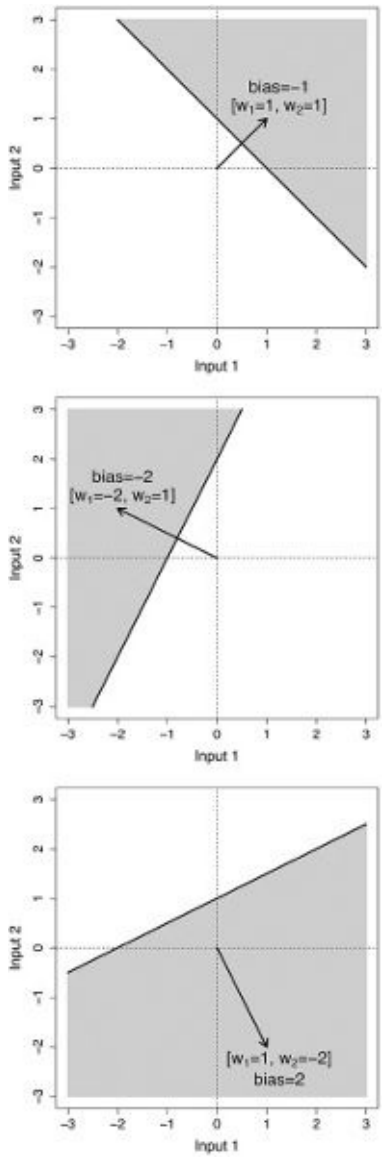


Figure 3.6 Decision boundary plots for a two-input neuron that illustrate the effect of the bias term on the decision boundary. Top: weight vector [w1=1, w2=1] and bias equal to -1; middle: weight vector [w1=-2, w2=1] and bias equal to -2; bottom: weight vector [w1=1, w2=-2] and bias equal to 2.

Instead of manually setting the value of the bias term, it is preferable to allow a neuron to learn the appropriate bias. The simplest way to do this is to treat the bias term as a weight and allow the neuron to learn the bias term at the same time that it is learning the rest of the weights for its inputs. All that is required to achieve this is to augment all the input vectors the neuron receives with an extra input that is always set to 1. By convention, this input is input 0 ($x_0 = 1$), and, consequently, the bias term is specified by weight 0 (w_0).[2] Figure 3.7 illustrates the structure of an artificial neuron when the bias term has been integrated as w_0.

When the bias term has been integrated into the weights of a neuron, the equation specifying the mapping from input(s) to output activation of the neuron can be simplified (at least from a notational perspective) as follows:

$$\text{Output} = \text{activation\_function}\left(z = \sum_{i=0}^{n} x_i \times w_i \right)$$

Notice that in this equation the index i goes from 0 to n, so that it now includes the fixed input, $x_0 = 1$, and the bias term, w_0; in the earlier version of this equation, the index only went from 1 to n. This new format means that the neuron is able to learn the bias term, simply by learning the appropriate weight w_0, using the same process that is used to learn the weights for the other inputs: at the start of training, the bias term for each neuron in the network will be initialized to a random value and then adjusted, along with the weights of the network, in response to the performance of the network on the dataset.

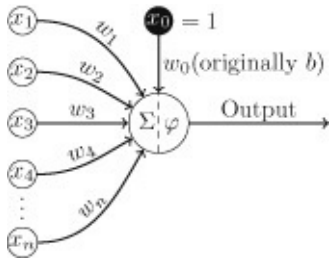


Figure 3.7 An artificial neuron with a bias term included as w_0.

Accelerating Neural Network Training Using GPUs

Merging the bias term is more than a notational convenience; it enables us to use specialized hardware to accelerate the training of neural networks. The fact that a bias term can be treated as the same as a weight means that the calculation of the weighted sum of inputs (including the addition of the bias term) can be treated as the multiplication of two vectors. As we discussed earlier, during the explanation of why the decision boundary was orthogonal to the weight vector, we can think of a set of inputs as a vector. Recognizing that much of the processing within a neural network involves vector and matrix multiplications opens up the possibility of using specialized hardware to speed up these calculations. For example, graphics processing units (GPUs) are hardware components that have specifically been designed to do extremely fast matrix multiplications.

In a standard feedforward network, all the neurons in one layer receive all the outputs (i.e., activations) from all the neurons in the preceding layer. This means that all the neurons in a layer receive the same set of inputs. As a result, we can calculate the weighted sum calculation for all the neurons in a layer using only a single vector by matrix multiplication. Doing this is much faster than calculating a separate weighted sum for each neuron in the layer. To do this calculation of weighted sums for an entire layer of neurons in a single multiplication, we put the outputs from the neurons in the preceding layer into a vector and store all the weights of the connections between the two layers of neurons in a matrix. We then multiply the vector by the matrix, and the resulting vector contains the weighted sums for all the neurons.

Figure 3.8 illustrates how the weighted summation calculations for all the neurons in a layer in a network can be calculated using a single matrix multiplication operation. This figure is composed of two separate graphics: the graphic on the left illustrates the connections between neurons in two layers of a network, and the graphic on the right illustrates the matrix operation to calculate the weighted sums for the neurons in the second layer of the network. To help maintain a correspondence between the two graphics, the connections into neuron E are highlighted in the graphic on the left, and the calculation of the weighted sum in neuron E is highlighted in the graphic on the right.

Focusing on the graphic on the right, the 1×3 vector (1 row, 3 columns) on the bottom-left of this graphic, stores the activations for the neurons in layer 1 of the network; note that these activations are the outputs from an activation

function φ (the particular activation function is not specified—it could be a threshold function, a tanh, a logistic function, or a rectified linear unit/ReLU function). The 3×4 matrix (three rows and four columns), in the top-right of the graphic, holds the weights for the connections between the two layers of neurons. In this matrix, each column stores the weights for the connections coming into one of the neurons in the second layer of the network. The first column stores the weights for neuron D, the second column for neuron E, etc.[3] Multiplying the 1×3 vector of activations from layer 1 by the 3×4 weight matrix results in a 1×4 vector corresponding to the weighted summations for the four neurons in layer 2 of the network: z_D is the weighted sum of inputs for neuron D, z_E for neuron E, and so on.

To generate the 1×4 vector containing the weighted summations for the neurons in layer 2, the activation vector is multiplied by each column in the matrix in turn. This is done by multiplying the first (leftmost) element in the vector by the first (topmost) element in the column, then multiplying the second element in the vector by the element in the second row in the column, and so on, until each element in the vector has been multiplied by its corresponding column element. Once all the multiplications between the vector and the column have been completed, the results are summed together and the stored in the output vector. Figure 3.8 illustrates multiplication of the activation vector by the second column in the weight matrix (the column containing the weights for inputs to neuron E) and the storing of the summation of these multiplications in the output vector as the value z_E.

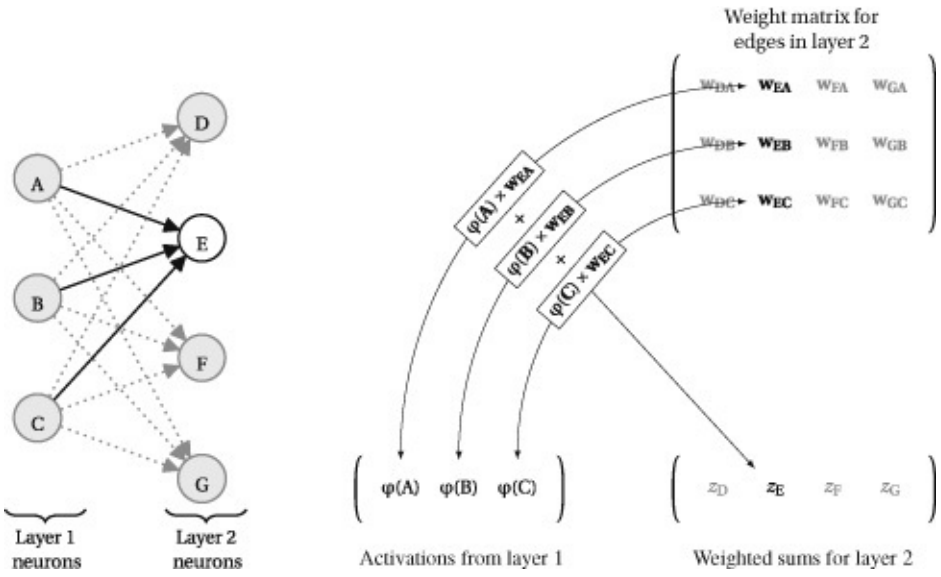


Figure 3.8 A graphical illustration of the topological connections of a specific neuron E in a network,

and the corresponding vector by matrix multiplication that calculates the weighted summation of inputs for the neuron E, and its siblings in the same layer.[5]

Indeed, the calculation implemented by an entire neural network can be represented as a chain of matrix multiplications, with an element-wise application of activation functions to the results of each multiplication. Figure 3.9 illustrates how a neural network can be represented in both graph form (on the left) and as a sequence of matrix operations (on the right). In the matrix representation, the $\times$ symbol represents standard matrix multiplication (described above) and the $\rightarrow \varphi \rightarrow$ notation represents the application of an activation function to each element in the vector created by the preceding matrix multiplication. The output of this element-wise application of the activation function is a vector containing the activations for the neurons in a layer of the network. To help show the correspondence between the two representations, both figures show the inputs to the network, I_1 and I_2, the activations from the three hidden units, A_1, A_2, and A_3, and the overall output of the network, y.

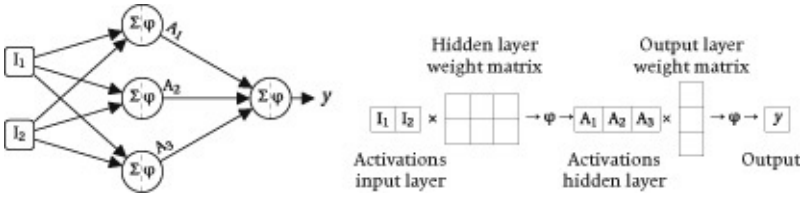


Figure 3.9 A graph representation of a neural network (left), and the same network represented as a sequence of matrix operations (right).[6]

As a side note, the matrix representation provides a transparent view of the depth of a network; the network's depth is counted as the number of layers that have a weight matrix associated with them (or equivalently, the depth of a network is the number of weight matrices required by the network). This is why the input layer is not counted when calculating the depth of a network: it does not have a weight matrix associated with it.

As mentioned above, the fact that the majority of calculations in a neural network can be represented as a sequence of matrix operations has important computational implications for deep learning. A neural network may contain over a million neurons, and the current trend is for the size of these networks to double every two to three years.[4] Furthermore, deep learning networks are trained by iteratively running a network on examples sampled from very large datasets and then updating the network parameters (i.e., the weights) to improve performance. Consequently, training a deep learning network can require very

large numbers of network runs, with each network run requiring millions of calculations. This is why computational speedups, such as those that can be achieved by using GPUs to perform matrix multiplications, have been so important for the development of deep learning.

The relationship between GPUs and deep learning is not one-way. The growth in demand for GPUs generated by deep learning has had a significant impact on GPU manufacturers. Deep learning has resulted in these companies refocusing their business. Traditionally, these companies would have focused on the computer games market, since the original motivation for developing GPU chips was to improve graphics rendering, and this had a natural application to computer games. However, in recent years these companies have focused on positioning GPUs as hardware for deep learning and artificial intelligence applications. Furthermore, GPU companies have also invested to ensure that their products support the top deep learning software frameworks.

Summary

The primary theme in this chapter has been that deep learning networks are composed of large numbers of simple processing units that work together to learn and implement complex mappings from large datasets. These simple units, neurons, execute a two-stage process: first, a weighted summation over the inputs to the neuron is calculated, and second, the result of the weighted summation is passed through a nonlinear function, known as an activation function. The fact that a weighted summation function can be efficiently calculated across a layer of neurons using a single matrix multiplication operation is important: it means that neural networks can be understood as a sequence of matrix operations; this has permitted the use of GPUs, hardware optimized to perform fast matrix multiplication, to speed up the training of networks, which in turn has enabled the size of networks to grow.

The compositional nature of neural networks means that it is possible to understand at a very fundamental level how a neural network operates. Providing a comprehensive description of this level of processing has been the focus of this chapter. However, the compositional nature of neural networks also raises a raft of questions in relation to how a network should be composed to solve a given task, for example:

- Which activation functions should the neurons in a network use?
- How many layers should there be in a network?
- How many neurons should there be in each layer?
- How should the neurons be connected together?

Unfortunately, many of these questions cannot be answered at a level of pure principle. In machine learning terminology, the types of concepts these questions are about are known as hyperparameters, as distinct from model parameters. The parameters of a neural network are the weights on the edges, and these are set by training the network using large datasets. By contrast, hyperparameters are the parameters of a model (in these cases, the parameters of a neural network architecture) and/or training algorithm that cannot be directly estimated from the data but instead must be specified by the person creating the model, either through the use of heuristic rules, intuition, or trial and error. Often, much of the effort that goes into the creation of a deep learning network involves experimental work to answer the questions in relation to hyperparameters, and

this process is known as hyperparameter tuning. The next chapter will review the history and evolution of deep learning, and the challenges posed by many of these questions are themes running through the review. Subsequent chapters in the book will explore how answering these questions in different ways can create networks with very different characteristics, each suited to different types of tasks. For example, recurrent neural networks are best suited to processing sequential/time-series data, whereas convolutional neural networks were originally developed to process images. Both of these network types are, however, built using the same fundamental processing unit, the artificial neuron; the differences in the behavior and abilities of these networks stems from how these neurons are arranged and composed.

4

A Brief History of Deep Learning

The history of deep learning can be described as three major periods of excitement and innovation, interspersed with periods of disillusionment. Figure 4.1 shows a timeline of this history, which highlights these periods of major research: on threshold logic units (early 1940s to the mid 1960s), connectionism (early 1980s to mid-1990s), and deep learning (mid 2000s to the present). Figure 4.1 distinguishes some of the primary characteristics of the networks developed in each of these three periods. The changes in these network characteristics highlight some of the major themes within the evolution of deep learning, including: the shift from binary to continuous values; the move from threshold activation functions, to logistic and tanh activation, and then onto ReLU activation; and the progressive deepening of the networks, from single layer, to multiple layer, and then onto deep networks. Finally, the upper half of figure 4.1 presents some of the important conceptual breakthroughs, training algorithms, and model architectures that have contributed to the evolution of deep learning.

Figure 4.1 provides a map of the structure of this chapter, with the sequence of concepts introduced in the chapter generally following the chronology of this timeline. The two gray rectangles in figure 4.1 represent the development of two important deep learning network architectures: convolutional neural networks (CNNs), and recurrent neural networks (RNNs). We will describe the evolution of these two network architectures in this chapter, and chapter 5 will give a more detailed explanation of how these networks work.

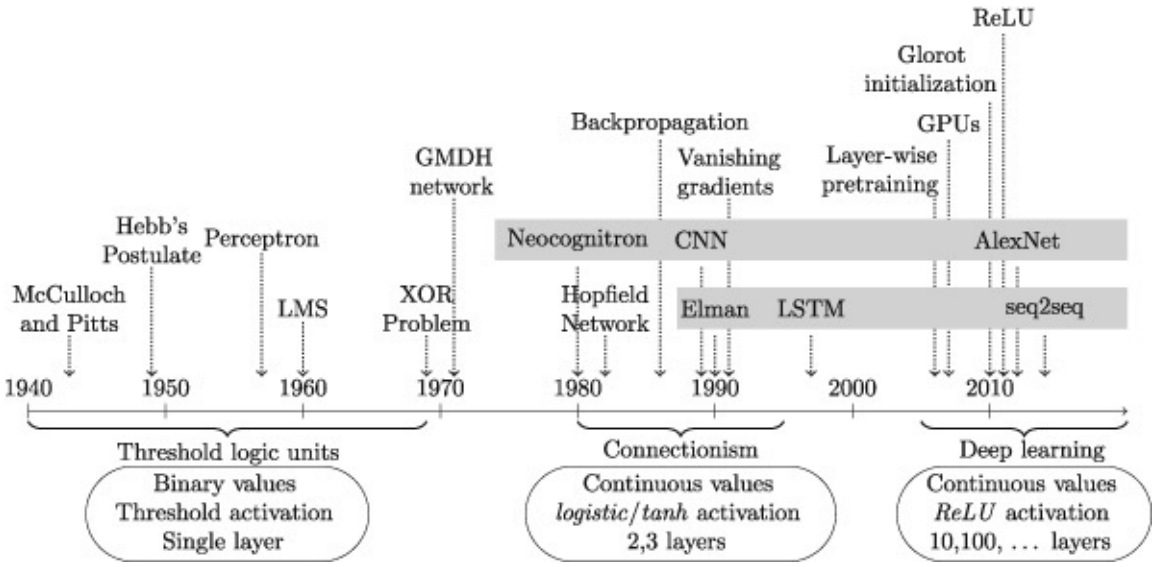


Figure 4.1 History of Deep Learning.

Early Research: Threshold Logic Units

In some of the literature on deep learning, the early neural network research is categorized as being part of cybernetics, a field of research that is concerned with developing computational models of control and learning in biological units. However, in figure 4.1, following the terminology used in Nilsson (1965), this early work is categorized as research on threshold logic units because this term transparently describes the main characteristics of the systems developed during this period. Most of the models developed in the 1940s, ’50s, and ’60s processed Boolean inputs (true/false represented as +1/-1 or 1/0) and generated Boolean outputs. They also used threshold activation functions (introduced in chapter 3), and were restricted to single-layer networks; in other words, they were restricted to a single matrix of tunable weights. Frequently, the focus of this early research was on understanding whether computational models based on artificial neurons had the capacity to learn logical relations, such as conjunction or disjunction.

In 1943, Walter McCulloch and Walter Pitts published an influential computational model of biological neurons in a paper entitled: “A Logical Calculus of the Ideas Immanent in Nervous Activity” (McCulloch and Pitts 1943). The paper highlighted the all-or-none characteristic of neural activity in the brain and set out to mathematically describe neural activity in terms of a calculus of propositional logic. In the McCulloch and Pitts model, all the inputs and the output to a neuron were either 0 or 1. Furthermore, each input was either excitatory (having a weight of +1) or inhibitory (having a weight of -1). A key concept introduced in the McCulloch and Pitts model was a summation of inputs followed by a threshold function being applied to the result of the summation. In the summation, if an excitatory input was on, it added 1; if an inhibitory input was on, it subtracted 1. If the result of the summation was above a preset threshold, then the output of the neuron was 1; otherwise, it output a 0. In the paper, McCulloch and Pitts demonstrated how logical operations (such as conjunction, disjunction, and negation) could be represented using this simple model. The McCulloch and Pitts model integrated the majority of the elements that are present in the artificial neurons introduced in chapter 3. In this model, however, the neuron was fixed; in other words the weights and threshold were set by han.

In 1949, Donald O. Hebb published a book entitled *The Organization of Behavior*, in which he set out a neuropsychological theory (integrating

psychology and the physiology of the brain) to explain general human behavior. The fundamental premise of the theory was that behavior emerged through the actions and interactions of neurons. For neural network research, the most important idea in this book was a postulate, now known as Hebb's postulate, which explained the creation of lasting memory in animals based on a process of changes to the connections between neurons:

> When an axon of a cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased. (Hebb 1949, p. 62)

This postulate was important because it asserted that information was stored in the connections between neurons (i.e., in the weights of a network), and furthermore that learning occurred by changing these connections based on repeated patterns of activation (i.e., learning can take place within a network by changing the weights of the network).

Rosenblatt's Perceptron Training Rule

In the years following Hebb's publication, a number of researchers proposed computational models of neuron activity that integrated the Boolean threshold activation units of McCulloch and Pitts, with a learning mechanism based on adjusting the weights applied to the inputs. The best known of these models was Frank Rosenblatt's perceptron model (Rosenblatt 1958). Conceptually, the perceptron model can be understood as a neural network consisting of a single artificial neuron that uses a threshold activation unit. Importantly, a perceptron network only has a single layer of weights. The first implementation of a perceptron was a software implementation on an IBM 704 system (and this was probably the first implementation of any neural network). However, Rosenblatt always intended the perceptron to be a physical machine and it was later implemented in custom-built hardware known as the "Mark 1 perceptron." The Mark 1 perceptron received input from a camera that generated a 400-pixel image that was passed into the machine via an array of 400 photocells that were in turn connected to the neurons. The weights on connections to the neurons were implemented using adjustable electrical resistors known as potentiometers, and weight adjustments were implemented by using electric motors to adjust the

potentiometers.

Rosenblatt proposed an error-correcting training procedure for updating the weights of a perceptron so that it could learn to distinguish between two classes of input: inputs for which the perceptron should produce the output $y = +1$, and inputs for which the perceptron should produce the output $y = -1$ (Rosenblatt 1960). The training procedure assumes a set of Boolean encoded input patterns, each with an associated target output. At the start of training, the weights in the perceptron are initialized to random values. Training then proceeds by iterating through the training examples, and after each example has been presented to the network, the weights of the network are updated based on the error between the output generated by the perceptron and the target output specified in the data. The training examples can be presented to the network in any order and examples may be presented multiple times before training is completed. A complete training pass through the set of examples is known as an iteration, and training terminates when the perceptron correctly classifies all the examples in an iteration.

Rosenblatt defined a learning rule (known as the perceptron training rule) to update each weight in a perceptron after a training example has been processed. The strategy the rule used to update the weights is the same as the three-condition strategy we introduced in chapter 2 to adjust the weights in the loan decision model:

1. If the output of the model for an example matches the output specified for that example in the dataset, then don't update the weights.
2. If the output of the model is too low for the current example, then increase the output of the model by increasing the weights for the inputs that had positive value for the example and decreasing the weights for the inputs that had a negative value for the example.
3. If the output of the model is too high for the current example, then reduce the output of the model by decreasing the weights for the inputs that had a positive value and increasing the weights for the inputs that had a negative value for the example.

Written out in an equation, Rosenblatt's learning rule updates a weight i (w_i) as:

$$w_i^{t+1} = w_i^t + \left(\eta \times \left(y^t - \hat{y}^t\right) \times x_i^t\right)$$

In this rule, w_i^{t+1} is the value of weight *i* after the network weights have been updated in response to the processing of example *t*, w_i^t is the value of weight *i* used during the processing of example *t*, η is a preset positive constant (known as the learning rate, discussed below), y^t is the expected output for example *t* as specified in the training dataset, $\hat{y}^t$ is the output generated by the perceptron for example *t*, and x_i^t is the component of input *t* that was weighted by w_i^t during the processing of the example.

Although it may look complex, the perceptron training rule is in fact just a mathematical specification of the three-condition weight update strategy described above. The primary part of the equation to understand is the calculation of the difference between the expected output and what the perceptron actually predicted: $y^t - \hat{y}^t$. The outcome of this subtraction tells us which of the three update conditions we are in. In understanding how this subtraction works, it is important to remember that for a perceptron model the desired output is always either $y = +1$ or $y = -1$. The first condition is when $y^t - \hat{y}^t = 0$; then the output of the perceptron is correct and the weights are not changed.

The second weight update condition is when the output of the perceptron is too large. This condition can only be occur when the correct output for example t is $y^t = -1$ and so this condition is triggered when $y^t - \hat{y}^t < 0$. In this case, if the perceptron output for the example t is $\hat{y}^t = +1$, then the error term is negative ($y^t - \hat{y}^t = -2$) and the weight w_i is updated by $+(\eta \times -2 \times x_i^t)$. Assuming, for the purpose of this explanation, that η is set to 0.5, then this weight update simplifies to $-x_i^t$. In other words, when the perceptron's output is too large, the weight update rule subtracts the input values from the weights. This will decrease the weights on inputs with positive values for the example, and increase the weights on inputs with negative values for the example (subtracting a negative number is the same as adding a positive number).

The third weight update condition is when the output of the perceptron is too small. This weight update condition is the exact opposite of the second. It can only occur when $y^t = +1$ and so is triggered when $y^t - \hat{y}^t > 0$. In this case ($y^t - \hat{y}^t = 2$), and the weight is updated by $+(\eta \times 2 \times x_i^t)$. Again assuming that η is set to 0.5, then this update simplifies to $+x_i^t$, which highlights that when the error

of the perceptron is positive, the rule updates the weight by adding the input to the weight. This has the effect of decreasing the weights on inputs with negative values for the example and increasing the weight on inputs with positive values for the example.

At a number of points in the preceding paragraphs we have referred to learning rate, η. The purpose of the learning rate, η, is to control the size of the adjustments that are applied to a weight. The learning rate is an example of a hyperparameter that is preset before the model is trained. There is a tradeoff in setting the learning rate:

- If the learning rate is too small, it may take a very long time for the training process to converge on an appropriate set of weights.
- If the learning rate is too large, the network's weights may jump around the weight space too much and the training may not converge at all.

One strategy for setting the learning rate is to set it to a relatively small positive value (e.g., 0.01), and another strategy is to initialize it to a larger value (e.g., 1.0) but to systematically reduce it as the training progresses

(e.g., $\eta^{t+1} = \eta^1 \times \frac{1}{t}$).

To make this discussion regarding the learning rate more concrete, imagine you are trying to solve a puzzle that requires you to get a small ball to roll into a hole. You are able to control the direction and speed of the ball by tilting the surface that the ball is rolling on. If you tilt the surface too steeply, the ball will move very fast and is likely to go past the hole, requiring you to adjust the surface again, and if you overadjust you may end up repeatedly tilting the surface. On the other hand, if you only tilt the surface a tiny bit, the ball may not start to move at all, or it may move very slowly taking a long time to reach the hole. Now, in many ways the challenge of getting the ball to roll into the hole is similar to the problem of finding the best set of weights for a network. Think of each point on the surface the ball is rolling across as a possible set of network weights. The ball's position at each point in time specifies the current set of weights of the network. The position of the hole specifies the optimal set of network weights for the task we are training the network to complete. In this context, guiding the network to the optimal set of weights is analogous to guiding the ball to the hole. The learning rate allows us to control how quickly we move across the surface as we search for the optimal set of weights. If we set

the learning rate to a high value, we move quickly across the surface: we allow large updates to the weights at each iteration, so there are big differences between the network weights in one iteration and the next. Or, using our rolling ball analogy, the ball is moving very quickly, and just like in the puzzle when the ball is rolling too fast and passes the hole, our search process may be moving so fast that it misses the optimal set of weights. Conversely, if we set the learning rate to a low value, we move very slowly across the surface: we only allow small updates to the weights at each iteration; or, in other words, we only allow the ball to move very slowly. With a low learning rate, we are less likely to miss the optimal set of weights, but it may take an inordinate amount of time to get to them. The strategy of starting with a high learning rate and then systematically reducing it is equivalent to steeply tilting the puzzle surface to get the ball moving and then reducing the tilt to control the ball as it approaches the hole.

Rosenblatt proved that if a set of weights exists that enables the perceptron to properly classify all of the training examples correctly, the perceptron training algorithm will eventually converge on this set of weights. This finding is known as the perceptron convergence theorem (Rosenblatt 1962). The difficulty with training a perceptron, however, is that it may require a substantial number of iterations through the data before the algorithm converges. Furthermore, for many problems it is unknown whether an appropriate set of weights exists in advance; consequently, if training has been going on for a long time, it is not possible to know whether the training process is simply taking a long time to converge on the weights and terminate, or whether it will never terminate.

The Least Mean Squares Algorithm

Around the same time that Rosenblatt was developing the perceptron, Bernard Widrow and Marcian Hoff were developing a very similar model called the ADALINE (short for adaptive linear neuron), along with a learning rule called the LMS (least mean square) algorithm (Widrow and Hoff 1960). An ADALINE network consists of a single neuron that is very similar to a perceptron; the only difference is that an ADALINE network does not use a threshold function. In fact, the output of an ADALINE network is the just the weighted sum of the inputs. This is why it is known as a linear neuron: a weighted sum is a linear function (it defines a line), and so an ADALINE network implements a linear mapping from inputs to output. The LMS rule is nearly identical to the perceptron learning rule, except that the output of the perceptron for a given example $\hat{y}^t$ is replaced by the weighted sum of the inputs:

$$w_i^{t+1} = w_i^t + \left(\eta \times \left(y^t - \left(\sum_{i=0}^{n} w_i^t \times x_i^t \right) \right) \times x_i^t \right)$$

The logic of the LMS update rule is the same as that of the perceptron training rule. If the output is too large, then weights that were applied to a positive input caused the output to be larger, and these weights should be decreased, and those that were applied to a negative input should be increased, thereby reducing the output the next time this input pattern is received. And, by the same logic, if the output is too small, then weights that were applied to a positive input are increased and those that were applied to a negative input should be decreased.

If the output of the model is too large, then weights associated with positive inputs should be reduced, whereas if the output is too small, then these weights should be increased.

One of the important aspects of Widrow and Hoff's work was to show that LMS rule could be used to train network to predict a number of any value, not just a +1 or -1. This learning rule was called the least mean square algorithm because using the LMS rule to iteratively adjust the weights in a neuron is equivalent to minimizing the average squared error on the training set. Today, the LMS learning rule is sometimes called the Widrow-Hoff learning rule, after the inventors; however, it is more commonly called the delta rule because it uses the difference (or delta) between desired output and the actual output to calculate the weight adjustments. In other words, the LMS rule specifies that a weight should be adjusted in proportion to the difference between the output of an ADALINE network and the desired output: if the neuron makes a large error, then the weights are adjusted by a large amount, if the neuron makes a small error, then weights are adjusted by a small amount.

Today, the perceptron is recognized as important milestone in the development of neural networks because it was the first neural network to be implemented. However, most modern algorithms for training neural networks are more similar to the LMS algorithm. The LMS algorithm attempts to minimize the mean squared error of the network. As will be discussed in chapter 6, technically this iterative error reduction process involves a gradient descent down an error surface; and, today, nearly all neural networks are trained using some variant of gradient descent.

The XOR Problem

The success of Rosenblatt, Widrow and Hoff, and others, in demonstrating that neural network models could automatically learn to distinguish between different sets of patterns, generated a lot of excitement around artificial intelligence and neural network research. However, in 1969, Marvin Minsky and Seymour Papert published a book entitled *Perceptrons*, which, in the annals of neural network research, is attributed with single-handedly destroying this early excitement and optimism (Minsky and Papert 1969). Admittedly, throughout the 1960s neural network research had suffered from a lot of hype, and a lack of success in terms of fulfilling the correspondingly high expectations. However, Minsky and Papert's book set out a very negative view of the representational power of neural networks, and after its publication funding for neural network research dried up.

Minsky and Papert's book primarily focused on single layer perceptrons. Remember that a single layer perceptron is the same as a single neuron that uses a threshold activation function, and so a single layer perceptron is restricted to implementing a linear (straight-line) decision boundary.[1] This means that a single layer perceptron can only learn to distinguish between two classes of inputs if it is possible to draw a straight line in the input space that has all of the examples of one class on one side of the line and all examples of the other class on the other side of the line. Minsky and Papert highlighted this restriction as a weakness of these models.

To understand Minsky and Papert's criticism of single layer perceptrons, we must first understand the concept of a linearly separable function. We will use a comparison between the logical AND and OR functions with the logical XOR function to explain the concept of a linearly separable function. The AND function takes two inputs, each of which can be either TRUE or FALSE, and returns TRUE if both inputs are TRUE. The plot on the left of figure 4.4 shows the input space for the AND function and categorizes each of the four possible input combinations as either resulting in an output value of TRUE (shown in the figure by using a clear dot) or FALSE (shown in the figure by using black dots). This plot illustrates that is possible to draw a straight line between the inputs for which the AND function returns TRUE, (T,T), and the inputs for which the function returns FALSE, {(F,F), (F,T), (T,F)}. The OR function is similar to the AND function, except that it returns TRUE if either or both inputs are TRUE. The middle plot in figure 4.4 shows that it is possible to draw a line that separates the inputs that the OR function classifies as TRUE, {(F,T), (T,F), (T,T)}, from those it classifies as FALSE, (F,F). It is because we can draw a

single straight line in the input space of these functions that divides the inputs belonging to one category of output from the inputs belonging to the other output category that the AND and OR functions are linearly separable functions.

The XOR function is also similar in structure to the AND and OR functions; however, it only returns TRUE if one (but not both) of its inputs are TRUE. The plot on the right of figure 4.2 shows the input space for the XOR function and categorizes each of the four possible input combinations as returning either TRUE (shown in the figure by using a clear dot) or FALSE (shown in the figure by using black dots). Looking at this plot you will see that it is not possible to draw a straight line between the inputs the XOR function classifies as TRUE and those that it classifies as FALSE. It is because we cannot use a single straight line to separate the inputs belonging to different categories of outputs for the XOR function that this function is said to be a nonlinearly separable function. The fact that the XOR function is nonlinearly separable does not make the function unique, or even rare—there are many functions that are nonlinearly separable.

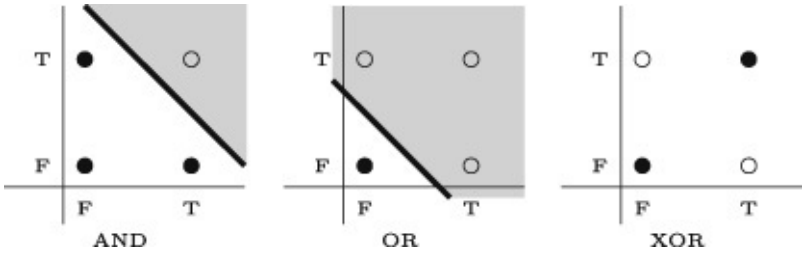


Figure 4.2 Illustrations of the linearly separable function. In each figure, black dots represent inputs for which the function returns FALSE, circles represent inputs for which the function returns TRUE. (T stands for true and F stands for false.)

The key criticism that Minsky and Papert made of single layer perceptrons was that these single layer models were unable to learn nonlinearly separable functions, such as the XOR function. The reason for this limitation is that the decision boundary of a perceptron is linear and so a single layer perceptron cannot learn to distinguish between the inputs that belong to one output category of a nonlinearly separable function from those that belong to the other category.

It was known at the time of Minsky and Papert's publication that it was possible to construct neural networks that defined a nonlinear decision boundary, and thus learn nonlinearly separable functions (such as the XOR function). The key to creating networks with more complex (nonlinear) decision boundaries was to extend the network to have multiple layers of neurons. For example, figure 4.3 shows a two-layer network that implements the XOR function. In this

network, the logical TRUE and FALSE values are mapped to numeric values: FALSE values are represented by 0, and TRUE values are represented by 1. In this network, units activate (output +1) if the weighted sum of inputs is ≥ 1; otherwise, they output 0. Notice that the units in the hidden layer implement the logical AND and OR functions. These can be understood as intermediate steps to solving the XOR challenge. The unit in the output layer implements the XOR by composing the outputs of these hidden layers. In other words, the unit in the output layer returns TRUE only when the AND node is off (output=0) and the OR node is on (output=1). However, it wasn't clear at the time how to train networks with multiple layers. Also, at the end of their book, Minsky and Papert argued that "in their judgment" the research on extending neural networks to multiple layers was "sterile" (Minsky and Papert 1969, sec. 13.2 page 23).

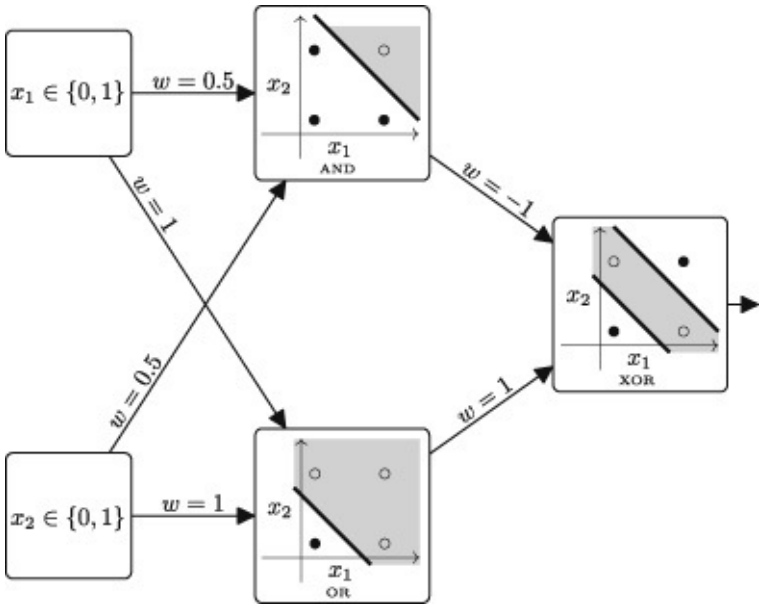


Figure 4.3 A network that implements the XOR function. All processing units use a threshold activation function with a threshold of ≥ 1.

In a somewhat ironic historical twist, contemporaneous with Minsky and Papert's publication, Alexey Ivakhnenko, a Ukrainian researcher, proposed the group method for data handling (GMDH), and in 1971 published a paper that described how it could be used to learn a neural network with eight layers (Ivakhnenko 1971). Today Ivakhnenko's 1971 GMDH network is credited with being the first published example of a deep network trained from data (Schmidhuber 2015). However, for many years, Ivaknenko's accomplishment was largely overlooked by the wider neural network community. As a consequence, very little of the current work in deep learning uses the GMDH

method for training: in the intervening years other training algorithms, such as backpropagation (described below), became standardized in the community. At the same time of Ivakhnenko's overlooked accomplishment, Minsky and Papert's critique was proving persuasive and it heralded the end of the first period of significant research on neural networks.

This first period of neural network research, did, however, leave a legacy that shaped the development of the field up to the present day. The basic internal structure of an artificial neuron was defined: a weighted sum of inputs fed through an activation function. The concept of storing information within the weights of a network was developed. Furthermore, learning algorithms based on iteratively adapting weights were proposed, along with practical learning rules, such as the LMS rule. In particular, the LMS approach, of adjusting the weights of neurons in proportion to the difference between the output of the neuron and the desired output, is present in most modern training algorithms. Finally, there was recognition of the limitations of single layer networks, and an understanding that one way to address these limitations was to extend the networks to include multiple layers of neurons. At this time, however, it was unclear how to train networks with multiple layers. Updating a weight requires an understanding of how the weight affects the error of the network. For example, in the LMS rule if the output of the neuron was too large, then weights that were applied to positive inputs caused the output to increase. Therefore, decreasing the size of these weight would reduce the output and thereby reduce the error. But, in the late 1960s, the question of how to model the relationship between the weights of the inputs to neurons in the hidden layers of a network and the overall error of the network was still unanswered; and, without this estimation of the contribution of the weight to the error, it was not possible to adjust the weights in the hidden layers of a network. The problem of attributing (or assigning) an amount of error to the components in a network is sometimes referred to as the credit assignment problem, or as the blame assignment problem.

Connectionism: Multilayer Perceptrons

In the 1980s, people began to reevaluate the criticisms of the late 1960s as being overly severe. Two developments, in particular, reinvigorated the field: (1) Hopfield networks; and (2) the backpropagation algorithm.

In 1982, John Hopfield published a paper where he described a network that could function as an associative memory (Hopfield 1982). During training, an associative memory learns a set of input patterns. Once the associate memory network has been trained, then, if a corrupted version of one of the input patterns is presented to the network, the network is able to regenerate the complete correct pattern. Associative memories are useful for a number of tasks, including pattern completion and error correction. Table 4.1[2] illustrates the tasks of pattern completion and error correction using the example of an associative memory that has been trained to store information on people's birthdays. In a Hopfield network, the memories, or input patterns, are encoded in binary strings; and, assuming binary patterns are relatively distinct from each other, a Hopfield network can store up to $0.138N$ of these strings, where N is the number of neurons in the network. So to store 10 distinct patterns requires a Hopfield network with 73 neurons, and to store 14 distinct patterns requires 100 neurons.

Table 4.1. Illustration of the uses of an association memory for pattern completion and error correction

Training patterns	**Pattern completion**		
`John**12May`	`Liz***?????`	→	`Liz***25Feb`
`Kerry*03Jan`	`???***10Mar`	→	`Des***10Mar`
`Liz***25Feb`	**Error correction**		
`Des***10Mar`	`Kerry*01Apr`	→	`Kerry*03Jan`
`Josef*13Dec`	`Jxsuf*13Dec`	→	`Josef*13Dec`

Backpropagation and Vanishing Gradients

In 1986, a group of researchers known as the parallel distributed processing (PDP) research group published a two-book overview of neural network research (Rumelhart et al. 1986b, 1986c). These books proved to be incredibly popular, and chapter 8 in volume one described the backpropagation algorithm (Rumelhart et al. 1986a). The backpropagation algorithm has been invented a number of times,[3] but it was this chapter by Rumelhart, Hinton, and Williams, published by PDP, that popularized its use. The backpropagation algorithm is a solution to the credit assignment problem and so it can be used to train a neural network that has hidden layers of neurons. The backpropagation algorithm is possibly the most important algorithm in deep learning. However, a clear and complete explanation of the backpropagation algorithm requires first explaining the concept of an error gradient, and then the gradient descent algorithm. Consequently, the in-depth explanation of backpropagation is postponed until chapter 6, which begins with an explanation of these necessary concepts. The general structure of the algorithm, however, can be described relatively quickly. The backpropagation algorithm starts by assigning random weights to each of the connections in the network. The algorithm then iteratively updates the weights in the network by showing training instances to the network and updating the network weights until the network is working as expected. The core algorithm works in a two-stage process. In the first stage (known as the forward pass), an input is presented to the network and the neuron activations are allowed to flow forward through the network until an output is generated. The second stage (known as the backward pass) begins at the output layer and works backward through the network until the input layer is reached. This backward pass begins by calculating an error for each neuron in the output layer. This error is then used to update the weights of these output neurons. Then the error of each output neuron is shared back (backpropagated) to the hidden neurons that connect to it, in proportion to the weights on the connections between the output neuron and the hidden neuron. Once this sharing (or blame assignment) has been completed for a hidden neuron, the total blame attributable to that hidden neuron is summed and this total is used to update the weights on that neuron. The backpropagation (or sharing back) of blame is then repeated for the neurons that have not yet had blame attributed to them. This process of blame assignment and weight updates continues back through the network until all the weights have been updated.

A key innovation that enabled the backpropagation algorithm to work was a change in the activation functions used in the neurons. The networks that were

developed in the early years of neural network research used threshold activation functions. The backpropagation algorithm does not work with threshold activation functions because backpropagation requires that the activation functions used by the neurons in the network be differentiable. Threshold activation functions are not differentiable because there is a discontinuity in the output of the function at the threshold. In other words, the slope of a threshold function at the threshold is infinite and therefore it is not possible to calculate the gradient of the function at that point. This led to the use of differentiable activation functions in multilayer neural networks, such as the logistic and tanh functions.

There is, however, an inherent limitation with using the backpropagation algorithm to train deep networks. In the 1980s, researchers found that backpropagation worked well with relatively shallow networks (one or two layers of hidden units), but that as the networks got deeper, the networks either took an inordinate amount of time to train, or else they entirely failed to converge on a good set of weights. In 1991, Sepp Hochreiter (working with Jürgen Schmidhuber) identified the cause of this problem in his diploma thesis (Hochreiter 1991). The problem is caused by the way the algorithm backpropagates errors. Fundamentally, the backpropagation algorithm is an implementation of the chain rule from calculus. The chain rule involves the multiplication of terms, and backpropagating an error from one neuron back to another can involve multiplying the error by a number terms with values less than 1. These multiplications by values less than 1 happen repeatedly as the error signal gets passed back through the network. This results in the error signal becoming smaller and smaller as it is backpropagated through the network. Indeed, the error signal often diminishes exponentially with respect to the distance from the output layer. The effect of this diminishing error is that the weights in the early layers of a deep network are often adjusted by only a tiny (or zero) amount during each training iteration. In other words, the early layers either train very, very slowly or do not move away from their random starting positions at all. However, the early layers in a neural network are vitally important to the success of the network, because it is the neurons in these layers that learn to detect the features in the input that the later layers of the network use as the fundamental building blocks of the representations that ultimately determine the output of the network. For technical reasons, which will be explained in chapter 6, the error signal that is backpropagated through the network is in fact the gradient of the error of the network, and, as a result, this problem of the error signal rapidly diminishing to near zero is known as the

vanishing gradient problem.

Connectionism and Local versus Distributed Representations

Despite the vanishing gradient problem, the backpropagation algorithm opened up the possibility of training more complex (deeper) neural network architectures. This aligned with the principle of connectionism. Connectionism is the idea that intelligent behavior can emerge from the interactions between large numbers of simple processing units. Another aspect of connectionism was the idea of a distributed representation. A distinction can be made in the representations used by neural networks between localist and distributed representations. In a localist representation there is a one-to-one correspondence between concepts and neurons, whereas in a distributed representation each concept is represented by a pattern of activations across a set of neurons. Consequently, in a distributed representation each concept is represented by the activation of multiple neurons and the activation of each neuron contributes to the representation of multiple concepts.

In a distributed representation each concept is represented by the activation of multiple neurons and the activation of each neuron contributes to the representation of multiple concepts.

To illustrate the distinction between localist and distributed representations, consider a scenario where (for some unspecified reason) a set of neuron activations is being used to represent the absence or presence of different foods. Furthermore, each food has two properties, the country of origin of the recipe and its taste. The possible countries of origin are: *Italy*, *Mexico*, or *France*; and, the set of possible tastes are: *Sweet*, *Sour*, or *Bitter*. So, in total there are nine possible types of food: *Italian+Sweet*, *Italian+Sour*, *Italian+Bitter*, *Mexican+Sweet*, etc. Using a localist representation would require nine neurons, one neuron per food type. There are, however, a number of ways to define a distributed representation of this domain. One approach is to assign a binary number to each combination. This representation would require only four neurons, with the activation pattern 0000 representing *Italian+Sweet*, 0001 representing *Italian+Sour*, 0010 representing *Italian+Bitter*, and so on up to 1000 representing *French+Bitter*. This is a very compact representation. However, notice that in this representation the activation of each neuron in isolation has no independently meaningful interpretation: the rightmost neuron

would be active (***1) for *Italian+Sour*, *Mexican+Sweet*, *Mexican+Bitter*, and *France+Sour*, and without knowledge of the activation of the other neurons, it is not possible know what country or taste is being represented. However, in a deep network the lack of semantic interpretability of the activations of hidden units is not a problem, so long as the neurons in the output layer of the network are able to combine these representations in such a way so as to generate the correct output. Another, more transparent, distributed representation of this food domain is to use three neurons to represent the countries and three neurons to represent the tastes. In this representation, the activation pattern 100100 could represent *Italian+Sweet*, 001100 could represent *French+Sweet*, and 001001 could represent *French+Bitter*. In this representation, the activation of each neuron can be independently interpreted; however the distribution of activations across the set of neurons is required in order to retrieve the full description of the food (country+taste). Notice, however, that both of these distributed representations are more compact than the localist representation. This compactness can significantly reduce the number of weights required in a network, and this in turn can result in faster training times for the network.

The concept of a distributed representation is very important within deep learning. Indeed, there is a good argument that deep learning might be more appropriately named representation learning—the argument being that the neurons in the hidden layers of a network are learning distributed representations of the input that are useful intermediate representations in the mapping from inputs to outputs that the network is attempting to learn. The task of the output layer of a network is then to learn how to combine these intermediate representations so as to generate the desired outputs. Consider again the network in figure 4.3 that implements the XOR function. The hidden units in this network learn an intermediate representation of the input, which can be understood as composed of the AND and OR functions; the output layer then combines this intermediate representation to generate the required output. In a deep network with multiple hidden layers, each subsequent hidden layer can be interpreted as learning a representation that is an abstraction over the outputs of the preceding layer. It is this sequential abstraction, through learning intermediate representations, that enables deep networks to learn such complex mappings from inputs to outputs.

Network Architectures: Convolutional and Recurrent Neural Networks

There are a considerable number of ways in which a set of neurons can be connected together. The network examples presented so far in the book have

been connected together in a relatively uncomplicated manner: neurons are organized into layers and each neuron in a layer is directly connected to all of the neurons in the next layer of the network. These networks are known as feedforward networks because there are no loops within the network connections: all the connections point forward from the input toward the output. Furthermore, all of our network examples thus far would be considered to be fully connected, because each neuron is connected to all the neurons in the next layer. It is possible, and often useful, to design and train networks that are not feedforward and/or that are not fully connected. When done correctly, tailoring network architectures can be understood as embedding into the network architecture information about the properties of the problem that the network is trying to learn to model.

A very successful example of incorporating domain knowledge into a network by tailoring the networks architecture is the design of convolutional neural networks (CNNs) for object recognition in images. In the 1960s, Hubel and Wiesel carried out a series of experiments on the visual cortex of cats (Hubel and Wiesel 1962, 1965). These experiments used electrodes inserted into the brains of sedated cats to study the response of the brain cells as the cats were presented with different visual stimuli. Examples of the stimuli used included bright spots or lines of light appearing at a location in the visual field, or moving across a region of the visual field. The experiments found that different cells responded to different stimuli at different locations in the visual field: in effect a single cell in the visual cortex would be wired to respond to a particular type of visual stimulus occurring within a particular region of the visual field. The region of the visual field that a cell responded to was known as the receptive field of the cell. Another outcome of these experiments was the differentiation between two types of cells: “simple” and “complex.” For simple cells, the location of the stimulus is critical with a slight displacement of the stimulus resulting in a significant reduction in the cell’s response. Complex cells, however, respond to their target stimuli regardless of where in the field of vision the stimulus occurs. Hubel and Wiesel (1965) proposed that complex cells behaved as if they received projections from a large number of simple cells all of which respond to the same visual stimuli but differing in the position of their receptive fields. This hierarchy of simple cells feeding into complex cells results in funneling of stimuli from large areas of the visual field, through a set of simple cells, into a single complex cell. Figure 4.4 illustrates this funneling effect. This figure shows a layer of simple cells each monitoring a receptive field at a different location in the visual field. The receptive field of the complex cell covers the layer of

simple cells, and this complex cell activates if any of the simple cells in its receptive field activates. In this way the complex cell can respond to a visual stimulus if it occurs at any location in the visual field.

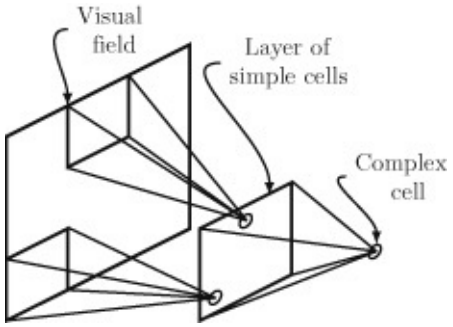


Figure 4.4 The funneling effect of receptive fields created by the hierarchy of simple and complex cells.

In the late 1970s and early 1980s, Kunihiko Fukushima was inspired by Hubel and Wiesel's analysis of the visual cortex and developed a neural network architecture for visual pattern recognition that was called the neocognitron (Fukushima 1980). The design of the neocognitron was based on the observation that an image recognition network should be able to recognize if a visual feature is present in an image irrespective of location in the image—or, to put it slightly more technically, the network should be able to do spatially invariant visual feature detection. For example, a face recognition network should be able to recognize the shape of an eye no matter where in the image it occurs, similar to the way a complex cell in Hubel and Wiesel's hierarchical model could detect the presence of a visual feature irrespective of where in the visual field it occurred.

Fukushima realized that the functioning of the simple cells in the Hubel and Wiesel hierarchy could be replicated in a neural network using a layer of neurons that all use the same set of weights, but with each neuron receiving inputs from fixed small regions (receptive fields) at different locations in the input field. To understand the relationship between neurons sharing weights and spatially invariant visual feature detection, imagine a neuron that receives a set of pixel values, sampled from a region of an image, as its inputs. The weights that this neuron applies to these pixel values define a visual feature detection function that returns true (high activation) if a particular visual feature (pattern) occurs in the input pixels, and false otherwise. Consequently, if a set of neurons all use the same weights, they will all implement the same visual feature detector. If the receptive fields of these neurons are then organized so that together they cover

the entire image, then if the visual feature occurs anywhere in the image at least one of the neurons in the group will identify it and activate.

Fukushima also recognized that the Hubel and Wiesel funneling effect (into complex cells) could be obtained by neurons in later layers also receiving as input the outputs from a fixed set of neurons in a small region of the preceding layer. In this way, the neurons in the last layer of the network each receive inputs from across the entire input field allowing the network to identify the presence of a visual feature anywhere in the visual input.

Some of the weights in neocognitron were set by hand, and others were set using an unsupervised training process. In this training process, each time an example is presented to the network a single layer of neurons that share the same weights is selected from the layers that yielded large outputs in response to the input. The weights of the neurons in the selected layer are updated so as to reinforce their response to that input pattern and the weights of neurons not in the layer are not updated. In 1989 Yann LeCun developed the convolutional neural network (CNN) architecture specifically for the task of image processing (LeCun 1989). The CNN architecture shared many of the design features found in the neocognitron; however, LeCun showed how these types of networks could be trained using backpropagation. CNNs have proved to be incredibly successful in image processing and other tasks. A particularly famous CNN is the AlexNet network, which won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in 2012 (Krizhevsky et al. 2012). The goal of the ILSVRC competition is to identify objects in photographs. The success of AlexNet at the ILSVRC competition generated a lot of excitement about CNNs, and since AlexNet a number of other CNN architectures have won the competition. CNNs are one of the most popular types of deep neural networks, and chapter 5 will provide a more detailed explanation of them.

Recurrent neural networks (RNNs) are another example of a neural network architecture that has been tailored to the specific characteristics of a domain. RNNs are designed to process sequential data, such as language. An RNN network processes a sequence of data (such as a sentence) one input at a time. An RNN has only a single hidden layer. However, the output from each of these hidden neurons is not only fed forward to the output neurons, it is also temporarily stored in a buffer and then fed back into all of the hidden neurons at the next input. Consequently, each time the network processes an input, each neuron in the hidden layer receives both the current input and the output the hidden layer generated in response to the previous input. In order to understand this explanation, it may at this point be helpful to briefly skip forward to figure

5.2 to see an illustration of the structure of an RNN and the flow of information through the network. This recurrent loop, of activations from the output of the hidden layer for one input being fed back into the hidden layer alongside the next input, gives an RNN a memory that enables it to process each input in the context of the previous inputs it has processed.[4] RNNs are considered deep networks because this evolving memory can be considered as deep as the sequence is long.

An early well-known RNN is the Elman network. In 1990, Jeffrey Locke Elman published a paper that described an RNN that had been trained to predict the endings of simple two- and three-word utterances (Elman 1990). The model was trained on a synthesized dataset of simple sentences generated using an artificial grammar. The grammar was built using a lexicon of twenty-three words, with each word assigned to a single lexical category (e.g., *man*=NOUN-HUM, *woman*=NOUN-HUM, *eat*=VERB-EAT, *cookie*=NOUN-FOOD, etc.). Using this lexicon, the grammar defined fifteen sentence generation templates (e.g., NOUN-HUM+VERB-EAT+NOUN-FOOD which would generate sentences such as *man eat cookie*). Once trained, the model was able to generate reasonable continuations for sentences, such as *woman+eat+? = cookie*. Furthermore, once the network was started, it was able to generate longer strings consisting of multiple sentences, using the context it generated itself as the input for the next word, as illustrated by this three-sentence example:

> *girl eat bread dog move mouse mouse move book*

Although this sentence generation task was applied to a very simple domain, the ability of the RNN to generate plausible sentences was taken as evidence that neural networks could model linguistic productivity without requiring explicit grammatical rules. Consequently, Elman's work had a huge impact on psycholinguistics and psychology. The following quote, from Churchland 1996, illustrates the importance that some researchers attributed to Elman's work:

> The productivity of this network is of course a feeble subset of the vast capacity that any normal English speaker commands. But productivity is productivity, and evidently a recurrent network can possess it. Elman's striking demonstration hardly settles the issue between the rule-centered approach to grammar and the network approach. That will be some time in working itself out. But the conflict is now an even one. I've made no secret where

my own bets will be placed. (Churchland 1996, p. 143)[5]

Although RNNs work well with sequential data, the vanishing gradient problem is particularly severe in these networks. In 1997, Sepp Hochreiter and Jürgen Schmidhuber, the researchers who in 1991 had presented an explanation of the vanishing gradient problem, proposed the long short-term memory (LSTM) units as a solution to this problem in RNNs (Hochreiter and Schmidhuber 1997). The name of these units draws on a distinction between how a neural network encodes long-term memory (understood as concepts that are learned over a period of time) through training and short-term memory (understood as the response of the system to immediate stimuli). In a neural network, long-term memory is encoded through adjusting the weights of the network and once trained these weights do not change. Short-term memory is encoded in a network through the activations that flow through the network and these activation values decay quickly. LSTM units are designed to enable the short-term memory (the activations) in the network to be propagated over long periods of time (or sequences of inputs). The internal structure of an LSTM is relatively complex, and we will describe it in chapter 5. The fact that LSTM can propagate activations over long periods enables them to process sequences that include long-distance dependencies (interactions between elements in a sequence that are separated by two or more positions). For example, the dependency between the subject and the verb in an English sentence: *The **dog/dogs** in that house **is/are** aggressive*. This has made LSTM networks suitable for language processing, and for a number of years they have been the default neural network architecture for many natural language processing models, including machine translation. For example, the sequence-to-sequence (seq2seq) machine translation architecture introduced in 2014 connects two LSTM networks in sequence (Sutskever et al. 2014). The first LSTM network, the encoder, processes the input sequence one input at a time, and generates a distributed representation of that input. The first LSTM network is called an encoder because it encodes the sequence of words into a distributed representation. The second LSTM network, the decoder, is initialized with the distributed representation of the input and is trained to generate the output sequence one element at a time using a feedback loop that feeds the most recent output element generated by the network back in as the input for the next time step. Today, this seq2seq architecture is the basis for most modern machine translation systems, and is explained in more detail in chapter 5.

By the late 1990s, most of the conceptual requirements for deep learning were

in place, including both the algorithms to train networks with multiple layers, and the network architectures that are still very popular today (CNNs and RNNs). However, the problem of the vanishing gradients still stifled the creation of deep networks. Also, from a commercial perspective, the 1990s (similar to the 1960s) experienced a wave of hype based on neural networks and unrealized promises. At the same time, a number of breakthroughs in other forms of machine learning models, such as the development of support vector machines (SVMs), redirected the focus of the machine learning research community away from neural networks: at the time SVMs were achieving similar accuracy to neural network models but were easier to train. Together these factors led to a decline in neural network research that lasted up until the emergence of deep learning.

The Era of Deep Learning

The first recorded use of the term deep learning is credited to Rina Dechter (1986), although in Dechter's paper the term was not used in relation to neural networks; and the first use of the term in relation to neural networks is credited to Aizenberg et al. (2000).[6] In the mid-2000s, interest in neural networks started to grow, and it was around this time that the term deep learning came to prominence to describe deep neural networks. The term deep learning is used to emphasize the fact that the networks being trained are much deeper than previous networks.

One of the early successes of this new era of neural network research was when Geoffrey Hinton and his colleagues demonstrated that it was possible to train a deep neural network using a process known as greedy layer-wise pretraining. Greedy layer-wise pretraining begins by training a single layer of neurons that receives input directly from the raw input. There are a number of different ways that this single layer of neurons can be trained, but one popular way is to use an autoencoder. An autoencoder is a neural network with three layers: an input layer, a hidden (encoding) layer, and an output (decoding) layer. The network is trained to reconstruct the inputs it receives in the output layer; in other words, the network is trained to output the exact same values that it received as input. A very important feature in these networks is that they are designed so that it is not possible for the network to simply copy the inputs to the outputs. For example, an autoencoder may have fewer neurons in the hidden layer than in the input and output layer. Because the autoencoder is trying to reconstruct the input at the output layer, the fact that the information from the input must pass through this bottleneck in the hidden layer forces the autoencoder to learn an encoding of the input data in the hidden layer that captures only the most important features in the input, and disregards redundant or superfluous information.[7]

Layer-Wise Pretraining Using Autoencoders

In layer-wise pretraining, the initial autoencoder learns an encoding for the raw inputs to the network. Once this encoding has been learned, the units in the hidden encoding layer are fixed, and the output (decoding) layer is thrown away. Then a second autoencoder is trained—but this autoencoder is trained to reconstruct the representation of the data generated by passing it through the

encoding layer of the initial autoencoder. In effect, this second autoencoder is stacked on top of the encoding layer of the first autoencoder. This stacking of encoding layers is considered to be a greedy process because each encoding layer is optimized independently of the later layers; in other words, each autoencoder focuses on finding the best solution for its immediate task (learning a useful encoding for the data it must reconstruct) rather than trying to find a solution to the overall problem for the network.

Once a sufficient number[8] of encoding layers have been trained, a tuning phase can be applied. In the tuning phase, a final network layer is trained to predict the target output for the network. Unlike the pretraining of the earlier layers of the network, the target output for the final layer is different from the input vector and is specified in the training dataset. The simplest tuning is where the pretrained layers are kept frozen (i.e., the weights in the pretrained layers don't change during the tuning); however, it is also feasible to train the entire network during the tuning phase. If the entire network is trained during tuning, then the layer-wise pretraining is best understood as finding useful initial weights for the earlier layers in the network. Also, it is not necessary that the final prediction model that is trained during tuning be a neural network. It is quite possible to take the representations of the data generated by the layer-wise pretraining and use it as the input representation for a completely different type of machine learning algorithm, for example, a support vector machine or a nearest neighbor algorithm. This scenario is a very transparent example of how neural networks learn useful representations of data prior to the final prediction task being learned. Strictly speaking, the term pretraining describes only the layer-wise training of the autoencoders; however, the term is often used to refer to both the layer-wise training stage and the tuning stage of the model.

Figure 4.5 shows the stages in layer-wise pretraining. The figure on the left illustrates the training of the initial autoencoder where an encoding layer (the black circles) of three units is attempting to learn a useful representation for the task of reconstructing an input vector of length 4. The figure in the middle of figure 4.5 shows the training of a second autoencoder stacked on top of the encoding layer of the first autoencoder. In this autoencoder, a hidden layer of two units is attempting to learn an encoding for an input vector of length 3 (which in turn is an encoding of a vector of length 4). The grey background in each figure demarcates the components in the network that are frozen during this training stage. The figure on the right shows the tuning phase where a final output layer is trained to predict the target feature for the model. For this example, in the tuning phase the pretrained layers in the network have been

frozen.

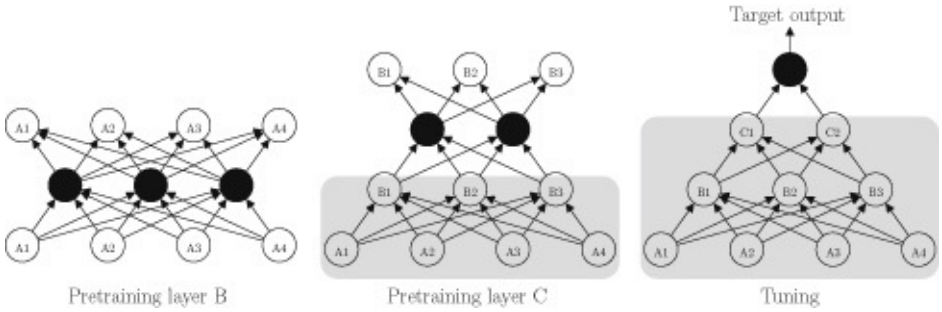


Figure 4.5 The pretraining and tuning stages in greedy layer-wise pretraining. Black circles represent the neurons whose training is the primary objective at each training stage. The gray background marks the components in the network that are frozen during each training stage.

Layer-wise pretraining was important in the evolution of deep learning because it was the first approach to training deep networks that was widely adopted.[9] However, today most deep learning networks are trained without using layer-wise pretraining. In the mid-2000s, researchers began to appreciate that the vanishing gradient problem was not a strict theoretical limit, but was instead a practical obstacle that could be overcome. The vanishing gradient problem does not cause the error gradients to disappear entirely; there are still gradients being backpropagated through the early layers of the network, it is just that they are very small. Today, there are a number of factors that have been identified as important in successfully training a deep network.

In the mid-2000s, researchers began to appreciate that the vanishing gradient problem was not a strict theoretical limit, but was instead a practical obstacle that could be overcome.

Weight Initialization and ReLU Activation Functions

One factor that is important in successfully training a deep network is how the network weights are initialized. The principles controlling how weight initialization affects the training of a network are still not clear. There are, however, weight initialization procedures that have been empirically shown to help with training a deep network. Glorot initialization[10] is a frequently used weight initialization procedure for deep networks. It is based on a number of assumptions but has empirical success to support its use. To get an intuitive understanding of Glorot initialization, consider the fact that there is typically a relationship between the magnitude of values in a set and the variance of the set:

generally the larger the values in a set, the larger the variance of the set. So, if the variance calculated on a set of gradients propagated through a layer at one point in the network is similar to the variance for the set of gradients propagated through another layer in a network, it is likely that the magnitude of the gradients propagated through both of these layers will also be similar. Furthermore, the variance of gradients in a layer can be related to the variance of the weights in the layer, so a potential strategy to maintain gradients flowing through a network is to ensure similar variances across each of the layer in a network. Glorot initialization is designed to initialize the weight in a network in such a way that all of the layers in a network will have a similar variance in terms of both forward pass activations and the gradients propagated during the backward pass in backpropagation. Glorot initialization defines a heuristic rule to meet this goal that involves sampling the weights for a network using the following uniform distribution (where w is the weight on a connection between layer j and j+i that is being initialized, U[-a,a] is the uniform distribution over the interval (-a,a), n_j is the number of neurons in layer j, and the notation w ~ U indicates that the value of w is sampled from distribution U)[11]:

$$w \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

Another factor that contributes to the success or failure of training a deep network is the selection of the activation function used in the neurons. Backpropagating an error gradient through a neuron involves multiplying the gradient by the value of the derivative of the activation function at the activation value of the neuron recorded during the forward pass. The derivatives of the logistic and tanh activation functions have a number of properties that can exacerbate the vanishing gradient problem if they are used in this multiplication step. Figure 4.6 presents a plot of the logistic function and the derivative of the logistic function. The maximum value of the derivative is 0.25. Consequently, after an error gradient has been multiplied by the value of the derivative of the logistic function at the appropriate activation for the neuron, the maximum value the gradient will have is a quarter of the gradient prior to the multiplication. Another problem with using the logistic function is that there are large portions of the domain of the function where the function is *saturated* (returning values that very close to 0 or 1), and the rate of change of the function in these regions is near zero; thus, the derivative of the function is near 0. This is an undesirable

property when backpropagating error gradients because the error gradients will be forced to zero (or close to zero) when backpropagated through any neuron whose activation is within one of these saturated regions. In 2011 it was shown that switching to a rectified linear activation function, $g(x) = \max(0, x)$, improved training for deep feedforward neural networks (Glorot et al. 2011). Neurons that use a rectified linear activation function are known as rectified linear units (ReLUs). One advantage of ReLUs is that the activation function is linear for the positive portion of its domain with a derivative equal to 1. This means that gradients can flow easily through ReLUs that have positive activation. However, the drawback of ReLUs is that the gradient of the function for the negative part of its domain is zero, so ReLUs do not train in this portion of the domain. Although undesirable, this is not necessarily a fatal flaw for learning because when backpropagating through a layer of ReLUs the gradients can still flow through the ReLUs in the layers that have positive activation. Furthermore, there are a number of variants of the basic ReLU that introduce a gradient on the negative side of the domain, a commonly used variant being the leaky ReLU (Maas et al. 2013). Today, ReLUs (or variants of ReLUs) are the most frequently used neurons in deep learning research.

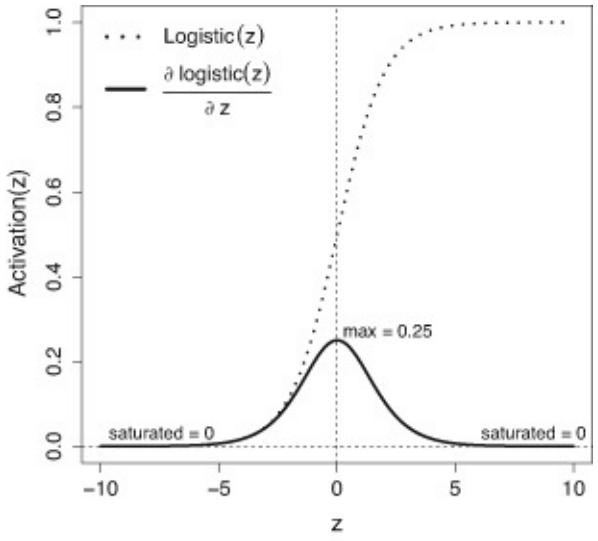


Figure 4.6 Plots of the logistic function and the derivative of the logistic function.

The Virtuous Cycle: Better Algorithms, Faster Hardware, Bigger Data

Although improved weight initialization methods and new activation functions have both contributed to the growth of deep learning, in recent years the two most important factors driving deep learning have been the speedup in computer

power and the massive increase in dataset sizes. From a computational perspective, a major breakthrough for deep learning occurred in the late 2000s with the adoption of graphical processing units (GPUs) by the deep learning community to speed up training. A neural network can be understood as a sequence of matrix multiplications that are interspersed with the application of nonlinear activation functions, and GPUs are optimized for very fast matrix multiplication. Consequently, GPUs are ideal hardware to speed up neural network training, and their use has made a significant contribution to the development of the field. In 2004, Oh and Jung reported a twentyfold performance increase using a GPU implementation of a neural network (Oh and Jung 2004), and the following year two further papers were published that demonstrated the potential of GPUs to speed up the training of neural networks: Steinkraus et al. (2005) used GPUs to train a two-layer neural network, and Chellapilla et al. (2006) used GPUs to train a CNN. However, at that time there were significant programming challenges to using GPUs for training networks (the training algorithm had to be implemented as a sequence of graphics operations), and so the initial adoption of GPUs by neural network researchers was relatively slow. These programming challenges were significantly reduced in 2007 when NVIDIA (a GPU manufacturer) released a C-like programming interface for GPUs called CUDA (compute unified device architecture).[12] CUDA was specifically designed to facilitate the use of GPUs for general computing tasks. In the years following the release of CUDA, the use of GPUs to speed up neural network training became standard.

However, even with these more powerful computer processors, deep learning would not have been possible unless massive datasets had also become available. The development of the internet and social media platforms, the proliferation of smartphones and “internet of things” sensors, has meant that the amount of data being captured has grown at an incredible rate over the last ten years. This has made it much easier for organizations to gather large datasets. This growth in data has been incredibly important to deep learning because neural network models scale well with larger data (and in fact they can struggle with smaller datasets). It has also prompted organizations to consider how this data can be used to drive the development of new applications and innovations. This in turn has driven a need for new (more complex) computational models in order to deliver these new applications. And, the combination of large data and more complex algorithms requires faster hardware in order to make the necessary computational workload tractable. Figure 4.7 illustrates the virtuous cycle between big data, algorithmic breakthroughs (e.g., better weight initialization,

ReLUs, etc.), and improved hardware that is driving the deep learning revolution.

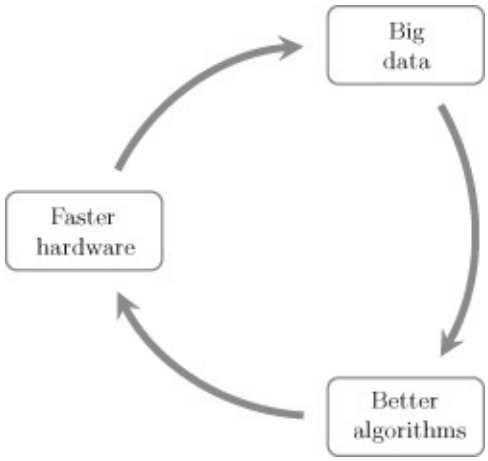


Figure 4.7 The virtuous cycle driving deep learning. Figure inspired by figure 1.2 in Reagen et al. 2017.

Summary

The history of deep learning reveals a number of underlying themes. There has been a shift from simple binary inputs to more complex continuous valued input. This trend toward more complex inputs is set to continue because deep learning models are most useful in high-dimensional domains, such as image processing and language. Images often have thousands of pixels in them, and language processing requires the ability represents and process hundreds of thousands of different words. This is why some of the best-known applications of deep learning are in these domains, for example, Facebook's face-recognition software, and Google's neural machine translation system. However, there are a growing number of new domains where large and complex digital datasets are being gathered. One area where deep learning has the potential to make a significant impact within the coming years is healthcare, and another complex domain is the sensor rich field of self-driving cars.

Somewhat surprisingly, at the core of these powerful models are simple information processing units: neurons. The connectionist idea that useful complex behavior can emerge from the interactions between large numbers of simple processing units is still valid today. This emergent behavior arises through the sequences of layers in a network learning a hierarchical abstraction of increasingly complex features. This hierarchical abstraction is achieved by each neuron learning a simple transformation of the input it receives. The network as a whole then composes these sequences of smaller transformations in order to apply a complex (highly) nonlinear mapping to the input. The output from the model is then generated by the final output layers of neuron, based the learned representation generated through the hierarchical abstraction. This is why depth is such an important factor in neural networks: the deeper the network, the more powerful the model becomes in terms of its ability to learn complex nonlinear mappings. In many domains, the relationship between input data and desired outputs involves just such complex nonlinear mappings, and it is in these domains that deep learning models outdo other machine learning approaches.

An important design choice in creating a neural network is deciding which activation function to use within the neurons in a network. The activation function within each neuron in a network is how nonlinearity is introduced into the network, and as a result it is a necessary component if the network is to learn a nonlinear mapping from inputs to output. As networks have evolved, so too

have the activation functions used in them. New activation functions have emerged throughout the history of deep learning, often driven by the need for functions with better properties for error-gradient propagation: a major factor in the shift from threshold to logistic and tanh activation functions was the need for differentiable functions in order to apply backpropagation; the more recent shift to ReLUs was, similarly, driven by the need to improve the flow of error gradients through the network. Research on activations functions is ongoing, and new functions will be developed and adopted in the coming years.

Another important design choice in creating a neural network is to decide on the structure of the network: for example, how should the neurons in the network be connected together? In the next chapter, we will discuss two very different answers to this question: convolution neural networks and recurrent neural networks.

5

Convolutional and Recurrent Neural Networks

Tailoring the structure of a network to the specific characteristics of the data from a task domain can reduce the training time of the network, and improves the accuracy of the network. Tailoring can be done in a number of ways, such as: constraining the connections between neurons in adjacent layers to subsets (rather than having fully connected layers); forcing neurons to share weights; or introducing backward connections into the network. Tailoring in these ways can be understood as building domain knowledge into the network. Another, related, perspective is it helps the network to learn by constraining the set of possible functions that it can learn, and by so doing guides the network to find a useful solution. It is not always clear how to fit a network structure to a domain, but for some domains where the data has a very regular structure (e.g., sequential data such as text, or gridlike data such as images) there are well-known network architectures that have proved successful. This chapter will introduce two of the most popular deep learning architectures: convolutional neural networks and recurrent neural networks.

Convolutional Neural Networks

Convolution neural networks (CNNs) were designed for image recognition tasks and were originally applied to the challenge of handwritten digit recognition (Fukushima 1980; LeCun 1989). The basic design goal of CNNs was to create a network where the neurons in the early layer of the network would extract local visual features, and neurons in later layers would combine these features to form higher-order features. A local visual feature is a feature whose extent is limited to a small patch, a set of neighboring pixels, in an image. For example, when applied to the task of face recognition, the neurons in the early layers of a CNN learn to activate in response to simple local features (such as lines at a particular angle, or segments of curves), neurons deeper in the network combine these low-level features into features that represent body parts (such as eyes or noises), and the neurons in the final layers of the network combine body part activations in order to be able to identify whole faces in an image.

Using this approach, the fundamental task in image recognition is learning the feature detection functions that can robustly identify the presence, or absence, of local visual features in an image. The process of learning functions is at the core of neural networks, and is achieved by learning the appropriate set of weights for the connections in the network. CNNs learn the feature detection functions for local visual features in this way. However, a related challenge is designing the architecture of the network so that the network will identify the presence of a local visual feature in an image irrespective of where in the image it occurs. In other words, the feature detection functions must be able to work in a translation invariant manner. For example, a face recognition system should be able to recognize the shape of an eye in an image whether the eye is in the center of the image or in the top-right corner of the image. This need for translation invariance has been a primary design principle of CNNs for image processing, as Yann LeCun stated in 1989:

> It seems useful to have a set of feature detectors that can detect a particular instance of a feature anywhere on the input plane. Since the precise location of a feature is not relevant to the classification, we can afford to lose some position information in the process. (LeCun 1989, p. 14)

CNNs achieve this translation invariance of local visual feature detection by using weight sharing between neurons. In an image recognition setting, the function implemented by a neuron can be understood as a visual feature detector. For example, neurons in the first hidden layer of the network will receive a set of pixel values as input and output a high activation if a particular pattern (local visual feature) is present in this set of pixels. The fact that the function implemented by a neuron is defined by the weights the neuron uses means that if two neurons use the same set of weights then they both implement the same function (feature detector). In chapter 4, we introduced the concept of a receptive field to describe the area that a neuron receives its input from. If two neurons share the same weights but have different receptive fields (i.e., each neuron inspects different areas of the input), then together the neurons act as a feature detector that activates if the feature occurs in either of the receptive fields. Consequently, it is possible to design a network with translation invariant feature detection by creating a set of neurons that share the same weights and that are organized so that: (1) each neuron inspects a different portion of the image; and (2) together the receptive fields of the neurons cover the entire image.

The scenario of searching an image in a dark room with a flashlight that has a narrow beam is sometimes used to explain how a CNN searches an image for local features. At each moment you can point the flashlight at a region of the image and inspect that local region. In this flashlight metaphor, the area of the image illuminated by the flashlight at any moment is equivalent to the receptive field of a single neuron, and so pointing the flashlight at a location is equivalent to applying the feature detection function to that local region. If, however, you want to be sure you inspect the whole image, then you might decide to be more systematic in how you direct the flashlight. For example, you might begin by pointing the flashlight at the top-left corner of the image and inspecting that region. You then move the flashlight to the right, across the image, inspecting each new location as it becomes visible, until you reach the right side of the image. You then point the flashlight back to the left of the image, but just below where you began, and move across the image again. You repeat this process until you reach the bottom-right corner of the image. The process of sequentially searching across an image and at each location in the search applying the same function to the local (illuminated) region is the essence of convolving a function across an image. Within a CNN, this sequential search across an image is implemented using a set of neurons that share weights and whose union of receptive fields covers the entire image.

Figure 5.1 illustrates the different stages of processing that are often found in

a CNN. The 6×6 matrix on the left of the figure represents the image that is the input to the CNN. The 4×4 matrix immediately to the right of the input represents a layer of neurons that together search the entire image for the presence of a particular local feature. Each neuron in this layer is connected to a different 3×3 receptive field (area) in the image, and they all apply the same weight matrix to their inputs:

$$\begin{bmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{bmatrix}$$

The receptive field of the neuron $[0,0]$ (top-left) in this layer is marked with the gray square covering the 3×3 area in the top-left of the input image. The dotted arrows emerging from each of the locations in this gray area represent the inputs to neuron $[0,0]$. The receptive field of the neighboring neuron $[0,1]$ is indicated by 3×3 square, outlined in bold in the input image. Notice that the receptive fields of these two neurons overlap. The amount of overlap of receptive fields is controlled by a hyperparameter called the stride length. In this instance, the stride length is one, meaning that for each position moved in the layer the receptive field of the neuron is translated by the same amount on the input. If the stride length hyperparameter is increased, the amount of overlap between receptive fields is decreased.

The receptive fields of both of these neurons ($[0,0]$ and $[0,1]$) are matrices of pixel values and the weights used by these neurons are also matrices. In computer vision, the matrix of weights applied to an input is known as the kernel (or convolution mask); the operation of sequentially passing a kernel across an image and within each local region, weighting each input and adding the result to its local neighbors, is known as a convolution. Notice that a convolution operation does not include a nonlinear activation function (this is applied at a later stage in processing). The kernel defines the feature detection function that all the neurons in the convolution implement. Convolving a kernel across an image is equivalent to passing a local visual feature detector across the image and recording all the locations in the image where the visual feature was present. The output from this process is a map of all the locations in the image where the relevant visual feature occurred. For this reason, the output of a convolution process is sometimes known as a feature map. As noted above, the convolution operation does not include a nonlinear activation function (it only involves a

weighted summation of the inputs). Consequently, it is standard to apply a nonlinearity operation to a feature map. Frequently, this is done by applying a rectified linear function to each position in a feature map; the rectified linear activation function is defined as: $\text{rectifier}(z) = \max(0,z)$. Passing a rectified linear activation function over a feature map simply changes all negative values to 0. In figure 5.1, the process of updating a feature map by applying a rectified linear activation function to each of its elements is represented by the layer labeled Nonlinearity.

The quote from Yann LeCun, at the start of this section, mentions that the precise location of a feature in an image may not be relevant to an image processing task. With this in mind, CNNs often discard location information in favor of generalizing the network's ability to do image classification. Typically, this is achieved by down-sampling the updated feature map using a pooling layer. In some ways pooling is similar to the convolution operation described above, in so far as pooling involves repeatedly applying the same function across an input space. For pooling, the input space is frequently a feature map whose elements have been updated using a rectified linear function. Furthermore, each pooling operation has a receptive field on the input space—although, for pooling, the receptive fields sometimes do not overlap. There are a number of different pooling functions used; the most common is called max pooling, which returns the maximum value of any of its inputs. Calculating the average value of the inputs is also used as a pooling function.

Convolving a kernel across an image is equivalent to passing a local visual feature detector across the image and recording all the locations in the image where the visual feature was present.

The operation sequence of applying a convolution, followed by a nonlinearity, to the feature map, and then down-sampling using pooling, is relatively standard across most CNNs. Often these three operations are together considered to define a convolutional layer in a network, and this is how they are presented in figure 5.1.

The fact that a convolution searches an entire image means that if the visual feature (pixel pattern) that the function (defined by shared kernel) detects occurs anywhere in the image, its presence will be recorded in the feature map (and if pooling is used, also in the subsequent output from the pooling layer). In this way, a CNN supports translation invariant visual feature detection. However, this

has the limitation that the convolution can only identify a single type of feature. CNNs generalize beyond one feature by training multiple convolutional layers in parallel (or filters), with each filter learning a single kernel matrix (feature detection function). Note the convolution layer in figure 5.1 illustrates a single filter. The outputs of multiple filters can be integrated in a variety of ways. One way to integrate information from different filters is to take the feature maps generated by the separate filters and combine them into a single multifilter feature map. A subsequent convolutional layer then takes this multifilter feature map as input. Another other way to integrate information from different filter is to use a densely connected layer of neurons. The final layer in figure 5.1 illustrates a dense layer. This dense layer operates in exactly the same way as a standard layer in a fully connected feedforward network. Each neuron in the dense layer is connected to all of the elements output by each of the filters, and each neuron learns a set of weights unique to itself that it applies to the inputs. This means that each neuron in a dense layer can learn a different way to integrate information from across the different filters.

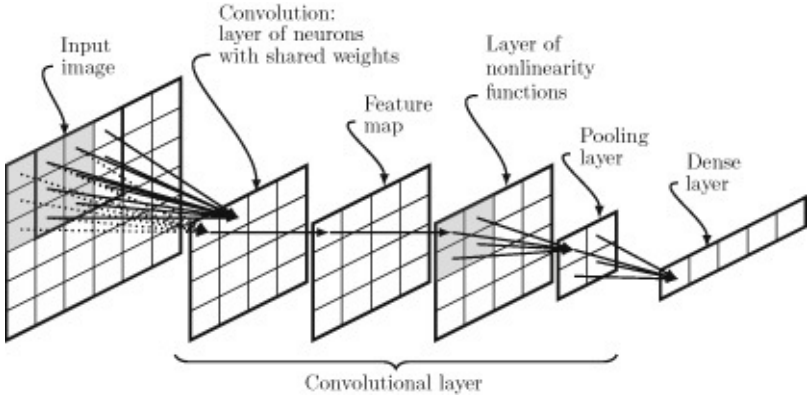


Figure 5.1 Illustrations of the different stages of processing in a convolutional layer. Note in this figure the Image and Feature Map are data structures; the other stages represent operations on data.

The AlexNet CNN, which won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in 2012, had five convolutional layers, followed by three dense layers. The first convolutional layer had ninety-six different kernels (or filters) and included a ReLU nonlinearity and pooling. The second convolution layer had 256 kernels and also included ReLU nonlinearity and pooling. The third, fourth, and fifth convolutional layers did not include a nonlinearity step or pooling, and had 384, 384, and 256 kernels, respectively. Following the fifth convolutional layer, the network had three dense layers with 4096 neurons each. In total, AlexNet had sixty million weights and 650,000 neurons. Although sixty million weights is a large number, the fact that many of the neurons shared

weights actually reduced the number of weights in the network. This reduction in the number of required weights is one of the advantages of CNN networks. In 2015, Microsoft Research developed a CNN network called ResNet, which won the ILSVRC 2015 challenge (He et al. 2016). The ResNet architecture extended the standard CNN architecture using skip-connections. A skip-connection takes the output from one layer in the network and feeds it directly into a layer that may be much deeper in the network. Using skip-connections it is possible to train very deep networks. In fact, the ResNet model developed by Microsoft Research had a depth of 152 layers.

Recurrent Neural Networks

Recurrent neural networks (RNNs) are tailored to the processing of sequential data. An RNN processes a sequence of data by processing each element in the sequence one at time. An RNN network only has a single hidden layer, but it also has a memory buffer that stores the output of this hidden layer for one input and feeds it back into the hidden layer along with the next input from the sequence. This recurrent flow of information means that the network processes each input within the context generated by processing the previous input, which in turn was processed in the context of the input preceding it. In this way, the information that flows through the recurrent loop encodes contextual information from (potentially) all of the preceding inputs in the sequence. This allows the network to maintain a memory of what it has seen previously in the sequence to help it decide what to do with the current input. The depth of an RNN arises from the fact that the memory vector is propagated forward and evolved through each input in the sequence; as a result an RNN network is considered as deep as a sequence is long.

The depth of an RNN arises from the fact that the memory vector is propagated forward and evolved through each input in the sequence; as a result an RNN network is considered as deep as a sequence is long.

Figure 5.2 illustrates the architecture of an RNN and shows how information flows through the network as it processes a sequence. At each time step, the network in this figure receives a vector containing two elements as input. The schematic on the left of figure 5.2 (time step=1.0) shows the flow of information in the network when it receives the first input in the sequence. This input vector is fed forward into the three neurons in the hidden layer of the network. At the same time these neurons also receive whatever information is stored in the memory buffer. Because this is the initial input, the memory buffer will only contain default initialization values. Each of the neurons in the hidden layer will process the input and generate an activation. The schematic in the middle of figure 5.2 (time step=1.5) shows how this activation flows on through the network: the activation of each neuron is passed to the output layer where it is processed to generate the output of the network, and it is also stored in the memory buffer (overwriting whatever information was stored there). The

elements of the memory buffer simply store the information written to them; they do not transform it in any way. As a result, there are no weights on the edges going from the hidden units to the buffer. There are, however, weights on all the other edges in the network, including those from the memory buffer units to the neurons in the hidden layer. At time step 2, the network receives the next input from the sequence, and this is passed to the hidden layer neurons along with the information stored in the buffer. This time the buffer contains the activations that were generated by the hidden neurons in response to the first input.

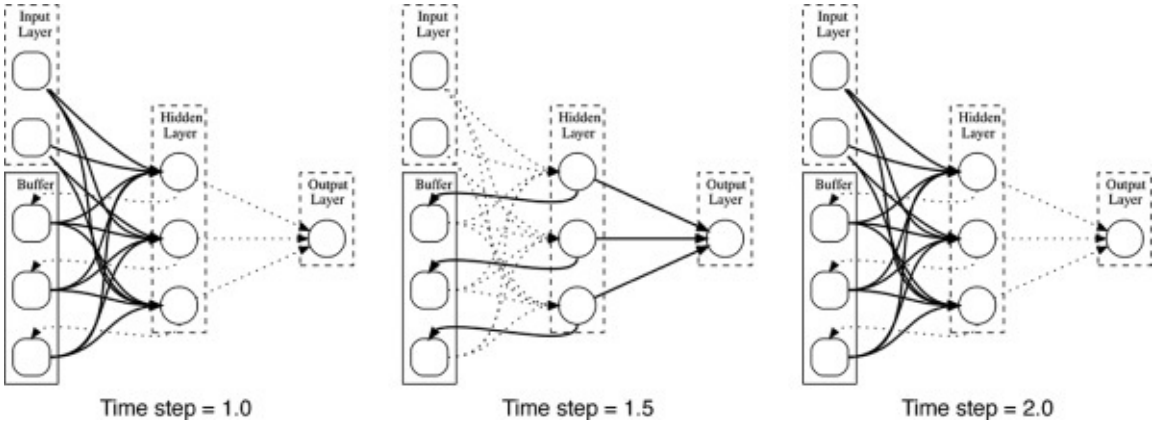


Figure 5.2 The flow of information in an RNN as it processes a sequence of inputs. The arrows in bold are the active paths of information flow at each time point; the dashed arrows show connections that are not active at that time.

Figure 5.3 shows an RNN that has been unrolled through time as it processes a sequence of inputs $[X_1, X_2, \ldots, X_t]$. Each box in this figure represents a layer of neurons. The box labeled h_0 represents the state of the memory buffer when the network is initialized; the boxes labeled $[h_1, \ldots, h_t]$ represent the hidden layer of the network at each time step; and the boxes labeled $[Y_1, \ldots, Y_t]$ represent the output layer of the network at each time step. Each of the arrows in the figure represents a set of connections between one layer and another layer. For example, the vertical arrow from X_1 to h_1 represents the connections between the input layer and the hidden layer at time step 1. Similarly, the horizontal arrows connecting the hidden layers represent the storing of the activations from a hidden state at one time step in the memory buffer (not shown) and the propagation of these activations to the hidden layer at the next time step through the connections from the memory buffer to the hidden state. At each time step, an input from the sequence is presented to the network and is fed forward to the

hidden layer. The hidden layer generates a vector of activations that is passed to the output layer and is also propagated forward to the next time step along the horizontal arrows connecting the hidden states.

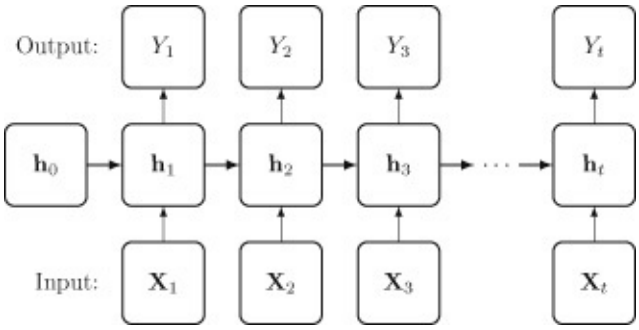


Figure 5.3 An RNN network unrolled through time as it processes a sequence of inputs $[X_1, X_2, ..., X_t]$.

Although RNNs can process a sequence of inputs, they struggle with the problem of vanishing gradients. This is because training an RNN to process a sequence of inputs requires the error to be backpropagated through the entire length of the sequence. For example, for the network in figure 5.3, the error calculated on the output Y_t must be backpropagated through the entire network so that it can be used to update the weights on the connections from h_0 and X_1 to h_1. This entails backpropagating the error through all the hidden layers, which in turn involves repeatedly multiplying the error by the weights on the connections feeding activations from one hidden layer forward to the next hidden layer. A particular problem with this process is that it is the same set of weights that are used on all the connections between the hidden layers: each horizontal arrow represents the same set of connections between the memory buffer and the hidden layer, and the weights on these connections are stationary through time (i.e., they don't change from one time step to the next during the processing of a given sequence of inputs). Consequently, backpropogating an error through *k* time steps involves (among other multiplications) multiplying the error gradient by the same set of weights *k* times. This is equivalent to multiplying each error gradient by a weight raised to the power of *k*. If this weight is less than 1, then when it is raised to a power, it diminishes at an exponential rate, and consequently, the error gradient also tends to diminish at an exponential rate with respect to the length of the sequence—and vanish.

Long short-term memory networks (LSTMs) are designed to reduce the effect of vanishing gradients by removing the repeated multiplication by the same weight vector during backpropagation in an RNN. At the core of an LSTM[1] unit

is a component called the cell. The cell is where the activation (the short-term memory) is stored and propagated forward. In fact, the cell often maintains a vector of activations. The propagation of the activations within the cell through time is controlled by three components called gates: the forget gate, the input gate, and the output gate. The forget gate is responsible for determining which activations in the cell should be forgotten at each time step, the input gate controls how the activations in the cell should be updated in response to the new input, and the output gate controls what activations should be used to generate the output in response to the current input. Each of the gates consists of layers of standard neurons, with one neuron in the layer per activation in the cell state.

Figure 5.4 illustrates the internal structure of an LSTM cell. Each of the arrows in this image represents a vector of activations. The cell runs along the top of the figure from left (c_{t-1}) to right (c_t). Activations in the cell can take values in the range -1 to +1. Stepping through the processing for a single input, the input vector x_t is first concatenated with the hidden state vector that has been propagated forward from the preceding time step h_{t-1}. Working from left to right through the processing of the gates, the forget gate takes the concatenation of the input and the hidden state and passes this vector through a layer of neurons that use a sigmoid (also known as logistic)[2] activation function. As a result of the neurons in the forget layer using sigmoid activation functions the output of this forget layer is a vector of values in the range 0 to 1. The cell state is then multiplied by this forget vector. The result of this multiplication is that activations in the cell state that are multiplied by components in the forget vector with values near 0 are forgotten, and activations that are multiplied by forget vector components with values near 1 are remembered. In effect, multiplying the cell state by the output of a sigmoid layer acts as a filter on the cell state.

Next, the input gate decides what information should be added to the cell state. The processing in this step is done by the components in the middle block of figure 5.4, marked Input. This processing is broken down into two subparts. First, the gate decides which elements in the cell state should be updated, and second it decides what information should be included in the update. The decision regarding which elements in the cell state should be updated is implemented using a similar filter mechanism to the forget gate: the concatenated input x_t plus hidden state h_{t-1} is passed through a layer of sigmoid units to generate a vector of elements, the same width as the cell, where each element in the vector is in the range 0 to 1; values near 0 indicate that the corresponding cell element will not be updated, and values near 1 indicate that the corresponding cell element will be updated. At the same time that the filter

vector is generated, the concatenated input and hidden state are also passed through a layer of tanh units (i.e., neurons that use the tanh activation function). Again, there is one tanh unit for each activation in the LSTM cell. This vector represents the information that may be added to the cell state. Tanh units are used to generate this update vector because tanh units output values in the range -1 to +1, and consequently the value of the activations in the cell elements can be both increased and decreased by an update.[3] Once these two vectors have been generated, the final update vector is calculated by multiplying the vector output from the tanh layer by the filter vector generated from the sigmoid layer. The resulting vector is then added to the cell using vector addition.

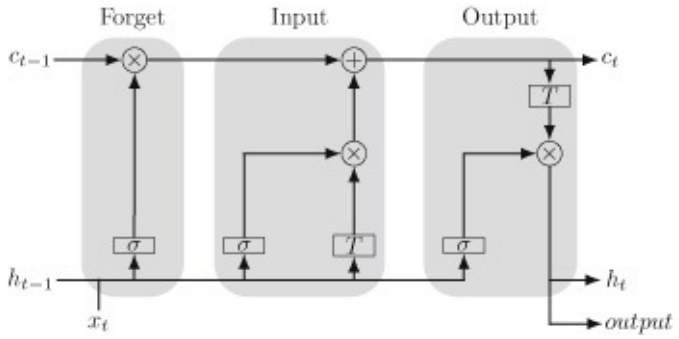


Figure 5.4 Schematic of the internal structure of an LSTM unit: σ represents a layer of neurons with sigmoid activations, $\tanh$ represents a layer of neurons with tanh activations, × represents vector multiplication, and + represents vector addition. The figure is inspired by an image by Christopher Olah available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

The final stage of processing in an LSTM is to decide which elements of the cell should be output in response to the current input. This processing is done by the components in the block marked Output (on the right of figure 5.4). A candidate output vector is generated by passing the cell through a tanh layer. At the same time, the concatenated input and propagated hidden state vector are passed through a layer of sigmoid units to create another filter vector. The actual output vector is then calculated by multiplying the candidate output vector by this filter vector. The resulting vector is then passed to the output layer, and is also propagated forward to the next time step as the new hidden state h_t.

The fact that an LSTM unit contains multiple layers of neurons means that an LSTM is a network in itself. However, an RNN can be constructed by treating an LSTM as the hidden layer in the RNN. In this configuration, an LSTM unit receives an input at each time step and generates an output for each input. RNNs that use LSTM units are often known as LSTM networks.

LSTM networks are ideally suited for natural language processing (NLP). A

key challenge in using a neural network to do natural language processing is that the words in language must be converted into vectors of numbers. The word2vec models, created by Tomas Mikolov and colleagues at Google research, are one of the most popular ways of doing this conversion (Mikolov et al. 2013). The word2vec models are based on the idea that words that appear in similar contexts have similar meanings. The definition of context here is surrounding words. So for example, the words *London* and *Paris* are semantically similar because each of them often co-occur with words that the other word also co-occurs with, such as: *capital*, *city*, *Europe*, *holiday*, *airport*, and so on. The word2vec models are neural networks that implement this idea of semantic similarity by initially assigning random vectors to each word and then using co-occurrences within a corpus to iteratively update these vectors so that semantically similar words end up with similar vectors. These vectors (known as word embeddings) are then used to represent a word when it is being input to a neural network.

One of the areas of NLP where deep learning has had a major impact is in machine translation. Figure 5.5 presents a high-level schematic of the seq2seq (or encoder-decoder) architecture for neural machine translation (Sutskever et al. 2014). This architecture is composed of two LSTM networks that have been joined together. The first LSTM network processes the input sentence in a word-by-word fashion. In this example, the source language is French. The words are entered into the system in reverse order as it has been found that this leads to better translations. The symbol eos is a special end of sentence symbol. As each word is entered, the encoder updates the hidden state and propagates it forward to the next time step. The hidden state generated by the encoder in response to the eos symbol is taken to be a vector representation of the input sentence. This vector is passed as the initial input to the decoder LSTM. The decoder is trained to output the translation sentence word by word, and after each word has been generated, this word is fed back into the system as the input for the next time step. In a way, the decoder is hallucinating the translation because it uses its own output to drive its own generation process. This process continues until the decoder outputs an eos symbol.

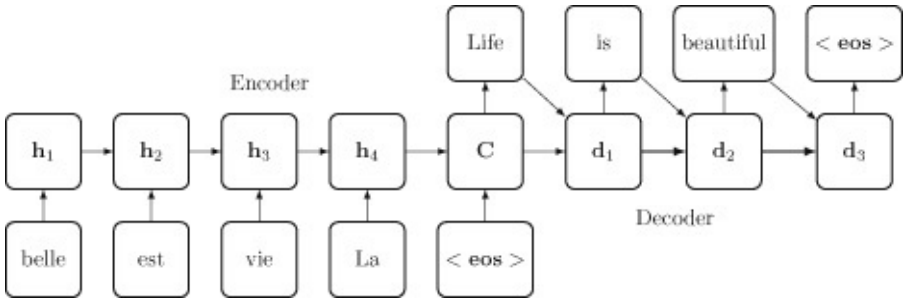


Figure 5.5 Schematic of the seq2seq (or encoder-decoder) architecture.

The idea of using a vector of numbers to represent the (interlingual) meaning of a sentence is very powerful, and this concept has been extended to the idea of using vectors to represent intermodal/multimodal representations. For example, an exciting development in recent years has been the development of automatic image captioning systems. These systems can take an image as input and generate a natural language description of the image. The basic structure of these systems is very similar to the neural machine translation architecture shown in figure 5.5. The main difference is that the encoder LSTM network is replaced by a CNN architecture that processes the input image and generates a vector representation that is then propagated to the decoder LSTM (Xu et al. 2015). This is another example of the power of deep learning arising from its ability to learn complex representations of information. In this instance, the system learns intermodal representations that enable information to flow from what is in an image to language. Combining CNN and RNN architectures is becoming more and more popular because it offers the potential to integrate the advantages of both systems and enables deep learning architectures to handle very complex data.

Irrespective of the network architecture we use, we need to find the correct weights for the network if we wish to create an accurate model. The weights of a neuron determine the transformation the neuron applies to its inputs. So, it is the weights of the network that define the fundamental building blocks of the representation the network learns. Today the standard method for finding these weights is an algorithm that came to prominence in the 1980s: backpropagation. The next chapter will present a comprehensive introduction to this algorithm.

6

Learning Functions

A neural network model, no matter how deep or complex, implements a function, a mapping from inputs to outputs. The function implemented by a network is determined by the weights the network uses. So, training a network (learning the function the network should implement) on data involves searching for the set of weights that best enable the network to model the patterns in the data. The most commonly used algorithm for learning patterns from data is the gradient descent algorithm. The gradient descent algorithm is very like the perceptron learning rule and the LMS algorithm described in chapter 4: it defines a rule to update the weights used in a function based on the error of the function. By itself the gradient descent algorithm can be used to train a single output neuron. However, it cannot be used to train a deep network with multiple hidden layers. This limitation is because of the credit assignment problem: how should the blame for the overall error of a network be shared out among the different neurons (including the hidden neurons) in the network? Consequently, training a deep neural network involves using both the gradient descent algorithm and the backpropagation algorithm in tandem.

The process used to train a deep neural network can be characterized as: randomly initializing the weight of a network, and then iteratively updating the weights of the network, in response to the errors the network makes on a dataset, until the network is working as expected. Within this training framework, the backpropagation algorithm solves the credit (or blame) assignment problem, and the gradient descent algorithm defines the learning rule that actually updates the weights in the network.

This chapter is the most mathematical chapter in the book. However, at a high level, all you need to know about the backpropagation algorithm and the gradient descent algorithm is that they can be used to train deep networks. So, if you don’t have the time to work through the details in this chapter, feel free to skim through it. If, however, you wish to get a deeper understanding of these two algorithms, then I encourage you to engage with the material. These algorithms are at the core of deep learning and understanding how they work is, possibly, the most direct way of understanding its potentials and limitations. I have attempted to present the material in this chapter in an accessible way, so if you are looking for a relatively gentle but still comprehensive introduction to these

algorithms, then I believe that this will provide it for you. The chapter begins by explaining the gradient descent algorithm, and then explains how gradient descent can be used in conjunction with the backpropagation algorithm to train a neural network.

Gradient Descent

A very simple type of function is a linear mapping from a single input to a single output. Table 6.1 presents a dataset with a single input feature and a single output. Figure 6.1 presents a scatterplot of this data along with a plot of the line that best fits this data. This line can be used as a function to map from an input value to a prediction of the output value. For example, if x = 0.9, then the response returned by this linear function is y = 0.6746. The error (or loss) of using this line as a model for the data is shown by the dashed lines from the line to each datum.

Table 6.1. A sample dataset with one input feature, x, and an output (target) feature, y

X	Y
0.72	0.54
0.45	0.56
0.23	0.38
0.76	0.57
0.14	0.17

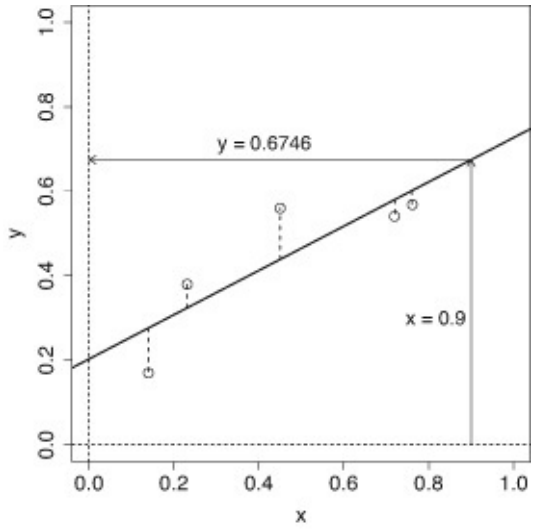


Figure 6.1 Scatterplot of data with "best fit" line and the errors of the line on each example plotted as vertical dashed line segments. The figure also shows the mapping defined by the line for input x=0.9 to output y=0.6746.

In chapter 2, we described how a linear function can be represented using the equation of a line:

$$y = mx + c$$

where m is the slope of the line, and c is the y-intercept, which specifies where the line crosses the y-axis. For the line in figure 6.1, $c = 0.203$ and $m = 0.524$; this is why the function returns the value $y = 0.6746$ when $x = 0.9$, as in the following:

$$0.6746 = (0.524 \times 0.9) + 0.203$$

The slope m and the y-intercept c are the parameters of this model, and these parameters can be varied to fit the model to the data.

The equation of a line has a close relationship with the weighted sum operation used in a neuron. This becomes apparent if we rewrite the equation of a line with model parameters rewritten as weights ($c \rightarrow w_0, m \rightarrow w_1$):

$$y = (w_0 \times 1) + (w_1 \times x)$$

Different lines (different linear models for the data) can be created by varying either of these weights (or model parameters). Figure 6.2 illustrates how a line

changes as the intercept and slope of the line varies: the dashed line illustrates what happens if the y-intercept is increased, and the dotted line shows what happens if the slope is decreased. Changing the y-intercept w_0 vertically translates the line, whereas modifying the slope w_1 rotates the line around the point $(x = 0, y = \text{intercept})$.

Each of these new lines defines a different function, mapping from x to y, and each function will have a different error with respect to how well it matches the data. Looking at figure 6.2, we can see that the full line, $[w_0 = 0.203, w_1 = 0.524]$, fits the data better than the other two lines because on average it passes closer to the data points. In other words, on average the error for this line for each data point is less than those of the other two lines. The total error of a model on a dataset can be measured by summing together the error the model makes on each example in the dataset. The standard way to calculate this total error is to use an equation known as the sum of squared errors (SSE):

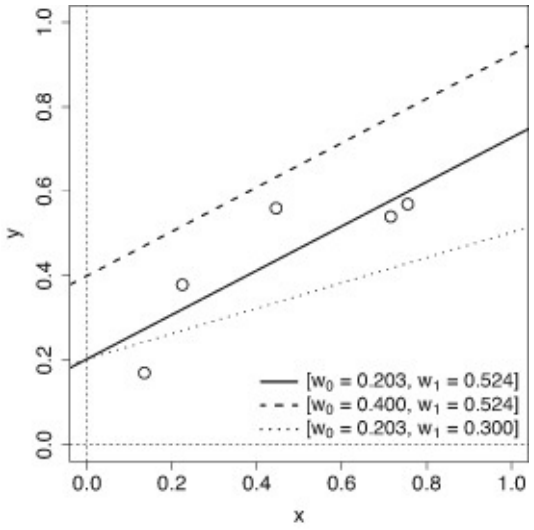


Figure 6.2 Plot illustrating how a line changes as the intercept (w_0) and slope (w_1) are varied.

$$SSE = \frac{1}{2}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2$$

This equation tells us how to add together the errors of a model on a dataset containing *n* examples. This equation calculates for each of the n examples in the dataset the error of the model by subtracting the prediction of the target value returned by the model from the correct target value for that example, as specified

in the dataset. In this equation y_j is the correct output value for target feature listed in the dataset for example *j*, and $\hat{y}_j$ is the estimate of the target value returned by the model for the same example. Each of these errors is then squared and these squared errors are then summed. Squaring the errors ensures that they are all positive, and therefore in the summation the errors for examples where the function underestimated the target do not cancel out the errors on examples where it overestimated the target. The multiplication of the summation of the errors by $1/2$, although not important for the current discussion, will become useful later. The lower the SSE of a function, the better the function models the data. Consequently, the sum of squared errors can be used as a fitness function to evaluate how well a candidate function (in this situation a model instantiating a line) matches the data.

Figure 6.3 shows how the error of a linear model varies as the parameters of the model change. These plots show the SSE of a linear model on the example single-input–single-output dataset listed in table 6.1. For each parameter there is a single best setting and as the parameter moves away from this setting (in either direction) the error of the model increases. A consequence of this is that the error profile of the model as each parameter varies is convex (bowl-shaped). This convex shape is particularly apparent in the top and middle plots in figure 6.3, which show that the SSE of the model is minimized when $w_0 = 0.203$ (lowest point of the curve in the top plot), and when $w_1 = 0.524$ (lowest point of the curve in the middle plot).

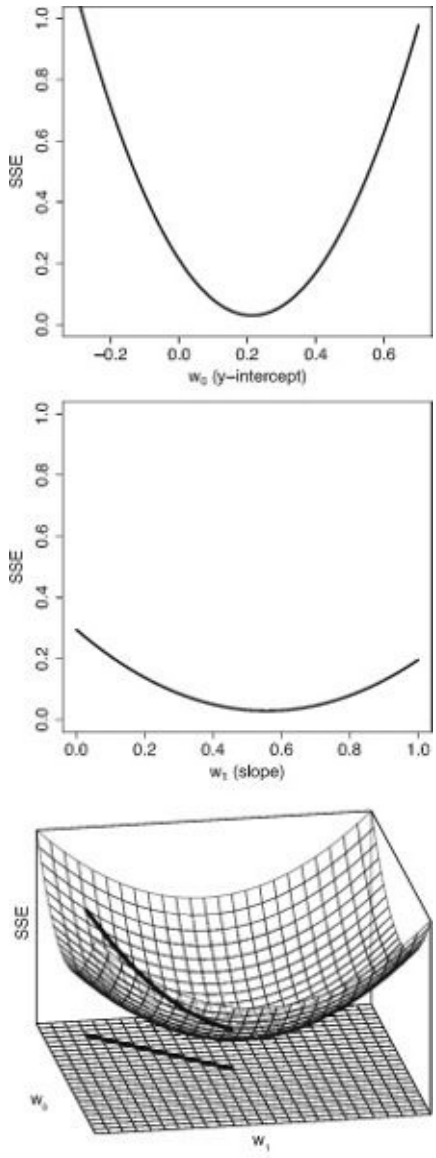


Figure 6.3 Plots of the changes in the error (SSE) of a linear model as the parameters of the model change. Top: the SSE profile of a linear model with a fixed slope $w_1 = 0.524$ when w_0 ranges across the interval 0.3 to 1. Middle: the SSE profile of a linear model with a y-intercept fixed at $w_0 = 0.203$ when w_1 ranges across the interval 0 to 1. Bottom: the error surface of the linear model when both w_0 and w_1 are varied.

If we plot the error of the model as both parameters are varied, we generate a three-dimensional convex bowl-shaped surface, known as an error surface. The bowl-shaped mesh in the plot at the bottom of figure 6.3 illustrates this error surface. This error surface was created by first defining a weight space. This weight space is represented by the flat grid at the bottom of the plot. Each coordinate in this weight space defines a different line because each coordinate

specifies an intercept (a w_0 value) and slope (a w_1 value). Consequently, moving across this planar weight space is equivalent to moving between different models. The second step in constructing the error surface is to associate an elevation with each line (i.e., coordinate) in the weight space. The elevation associated with each weight space coordinate is the SSE of the model defined by that coordinate; or, put more directly, the height of the error surface above the weight space plane is the SSE of the corresponding linear model when it is used as a model for the dataset. The weight space coordinates that correspond with the lowest point of the error surface define the linear model that has the lowest SSE on the dataset (i.e., the linear model that best fits the data).

The shape of the error surface in the plot on the right of figure 6.3 indicates that there is only a single best linear model for this dataset because there is a single point at the bottom of the bowl that has a lower elevation (lower error) than any other points on the surface. Moving away from this best model (by varying the weights of the model) necessarily involves moving to a model with a higher SSE. Such a move is equivalent to moving to a new coordinate in the weight space, which has a higher elevation associated with it on the error surface. A convex or bowl-shaped error surface is incredibly useful for learning a linear function to model a dataset because it means that the learning process can be framed as a search for the lowest point on the error surface. The standard algorithm used to find this lowest point is known as gradient descent.

A convex or bowl-shaped error surface is incredibly useful for learning a linear function to model a dataset because it means that the learning process can be framed as a search for the lowest point on the error surface.

The gradient descent algorithm begins by creating an initial model using a randomly selected a set of weights. Next the SSE of this randomly initialized model is calculated. Taken together, the guessed set of weights and the SSE of the corresponding model define the initial starting point on the error surface for the search. It is very likely that the randomly initialized model will be a bad model, so it is very likely that the search will begin at a location that has a high elevation on the error surface. This bad start, however, is not a problem, because once the search process is positioned on the error surface, the process can find a better set of weights by simply following the gradient of the error surface downhill until it reaches the bottom of the error surface (the location where moving in any direction results in an increase in SSE). This is why the algorithm is known as gradient descent: the gradient that the algorithm descends is the

gradient of the error surface of the model with respect to the data.

An important point is that the search does not progress from the starting location to the valley floor in one weight update. Instead, it moves toward the bottom of the error surface in an iterative manner, and during each iteration the current set of weights are updated so as to move to a nearby location in the weight space that has a lower SSE. Reaching the bottom of the error surface can take a large number of iterations. An intuitive way of understanding the process is to imagine a hiker who is caught on the side of a hill when a thick fog descends. Their car is parked at the bottom of the valley; however, due to the fog they can only see a few feet in any direction. Assuming that the valley has a nice convex shape to it, they can still find their way to their car, despite the fog, by repeatedly taking small steps that move down the hill following the local gradient at the position they are currently located. A single run of a gradient descent search is illustrated in the bottom plot of figure 6.3. The black curve plotted on the error surface illustrates the path the search followed down the surface, and the black line on the weight space plots the corresponding weight updates that occurred during the journey down the error surface. Technically, the gradient descent algorithm is known as an optimization algorithm because the goal of the algorithm is to find the optimal set of weights.

The most important component of the gradient descent algorithm is the rule that defines how the weights are updated during each iteration of the algorithm. In order to understand how this rule is defined it is first necessary to understand that the error surface is made up of multiple error gradients. For our simple example, the error surface is created by combining two error curves. One error curve is defined by the changes in the SSE as w_0 changes, shown in the top plot of figure 6.3. The other error curve is defined by the changes in the SSE as w_1 changes, shown in the plot in the middle of figure 6.3. Notice that the gradient of each of these curves can vary along the curve, for example, the w_0 error curve has a steep gradient on the extreme left and right of the plot, but the gradient becomes somewhat shallower in the middle of the curve. Also, the gradients of two different curves can vary dramatically; in this particular example the w_0 error curve generally has a much steeper gradient than the w_1 error curve.

The fact that the error surface is composed of multiple curves, each with a different gradient, is important because the gradient descent algorithm moves down the combined error surface by independently updating each weight so as to move down the error curve associated with that weight. In other words, during a single iteration of the gradient descent algorithm, w_0 is updated to move down

the w_0 error curve and w_1 is updated the move down the w_1 error curve. Furthermore, the amount each weight is updated in an iteration is proportional to the steepness of the gradient of the weight's error curve, and this gradient will vary from one iteration to the next as the process moves down the error curve. For example, w_0 will be updated by relatively large amounts in iterations where the search process is located high up on either side of the w_0 error curve, but by smaller amounts in iterations where the search process is nearer to the bottom of the w_0 error curve.

The error curve associated with each weight is defined by how the SSE changes with respect to the change in the value of the weight. Calculus, and in particular differentiation, is the field of mathematics that deals with rates of change. For example, taking the derivative of a function, $y = f(x)$, calculates the rate of change of y (the output) for each unit change in x (the input). Furthermore, if a function takes multiple inputs [$y = f(x_1,\ldots,x_n)$] then it is possible to calculate the rate of change of the output, y, with respect to changes in each of these inputs, x_i, by taking the partial derivative of the function of with respect to each input. The partial derivative of a function with respect to a particular input is calculated by first assuming that all the other inputs are held constant (and so their rate of change is 0 and they disappear from the calculation) and then taking the derivative of what remains. Finally, the rate of change of a function for a given input is also known as the gradient of the function at the location on the curve (defined by the function) that is specified by the input. Consequently, the partial derivative of the SSE with respect to a weight specifies how the output of the SSE changes as that weight changes, and so it specifies the gradient of the error curve of the weight. This is exactly what is needed to define the gradient descent weight update rule: the partial derivative of the SSE with respect to a weight specifies how to calculate the gradient of the weight's error curve, and in turn this gradient specifies how the weight should be updated to reduce the error (the output of the SSE).

The partial derivative of a function with respect to a particular variable is the derivative of the function when all the other variables are held constant. As a result there is a different partial derivative of a function with respect to each variable, because a different set of terms are considered constant in the calculation of each of the partial derivatives. Therefore, there is a different partial derivative of the SSE for each weight, although they all have a similar form. This is why each of the weights is updated independently in the gradient descent algorithm: the weight update rule is dependent on the partial derivative

of the SSE for each weight, and because there is a different partial derivative for each weight, there is a separate weight update rule for each weight. Again, although the partial derivative for each weight is distinct, all of these derivatives have the same form, and so the weight update rule for each weight will also have the same form. This simplifies the definition of the gradient descent algorithm. Another simplifying factor is that the SSE is defined relative to a dataset with n examples. The relevance of this is that the only variables in the SSE are the weights; the target output y and the inputs x are all specified by the dataset for each example, and so can be considered constants. As a result, when calculating the partial derivative of the SSE with respect to a weight, many of the terms in the equation that do not include the weight can be deleted because they are considered constants.

The relationship between the output of the SSE and each weight becomes more explicit if the SSE definition is rewritten so that the term $\hat{y}_j$, denoting the output predicted by the model, is replaced by the structure of the model generating the prediction. For the model with a single input x_1 and a dummy input, $x_0 = 1$, this rewritten version of the SSE is:

$$SSE = \frac{1}{2}\sum_{j=1}^{n}\left(y_j - \left(w_0 \times x_{j,0} + w_1 \times x_{j,1}\right)\right)^2$$

This equation uses a double subscript on the inputs, the first subscript j identifies the example (or row in the dataset) and the second subscript specifies the feature (or column in the dataset) of the input. For example, $x_{j,1}$ represents feature 1 from example j. This definition of the SSE can be generalized to a model with m inputs:

$$SSE = \frac{1}{2}\sum_{j=1}^{n}\left(y_j - \left(\sum_{i=0}^{m} w_i \times x_{j,i}\right)\right)^2$$

Calculating the partial derivative of the SSE with respect to a specific weight involves the application of the chain rule from calculus and a number of standard differentiation rules. The result of this derivation is the following equation (for simplicity of presentation we switch back to the notation $\hat{y}_i$ to represent the output from the model):

$$\frac{\partial\, \text{SSE}}{\partial\, w_i} = \sum_{j=1}^{n} \left(\underbrace{(y_j - \hat{y}_j)}_{\substack{\text{error of the} \\ \text{output of the} \\ \text{weighted sum}}} \times \underbrace{-x_{j,i}}_{\substack{\text{rate of change of} \\ \text{weighted sum} \\ \text{with respect to} \\ \text{change in } w_i}} \right)$$

This partial derivative specifies how to calculate the error gradient for weight w_i for the dataset where $x_{j,i}$ is the input associated with w_i for each example in the dataset. This calculation involves multiplying two terms, the error of the output and the rate of change of the output (i.e., the weighted sum) with respect to changes in the weight. One way of understanding this calculation is that if changing the weight changes the output of the weighted sum by a large amount, then the gradient of the error with respect to the weight is large (steep) because changing the weight will result in big changes in the error. However, this gradient is the uphill gradient, and we wish to move the weights so as to move down the error curve. So in the gradient descent weight update rule (shown below) the "–" sign in front of the input $x_{j,i}$ is dropped. Using t to represent the iteration of the algorithm (an iteration involves a single pass through the n examples in the dataset), the gradient descent weight update rule is defined as:

$$w_i^{t+1} = w_i^t + \left(\eta \times \underbrace{\sum_{j=1}^{n} \left((y_j^t - \hat{y}_j^t) \times x_{j,i}^t \right)}_{\text{error gradient for } w_i} \right)$$

There are a number of notable factors about this weight update rule. First, the rule specifies how the weight w_i should be updated after iteration t through the dataset. This update is proportional to the gradient of the error curve for the weight for that iteration (i.e., the summation term, which actually defines the partial derivative of the SSE for that weight). Second, the weight update rule can be used to update the weights for functions with multiple inputs. This means that the gradient descent algorithm can be used to descend error surfaces with more than two weight coordinates. It is not possible to visualize these error surfaces because they will have more than three dimensions, but the basic principles of descending an error surface using the error gradient generalizes to learning

functions with multiple inputs. Third, although the weight update rule has a similar structure for each weight, the rule does define a different update for each weight during each iteration because the update is dependent on the inputs in the dataset examples to which the weight is applied. Fourth, the summation in the rule indicates that, in each iteration of the gradient descent algorithm, the current model should be applied to all n of the examples in the dataset. This is one of the reasons why training a deep learning network is such a computationally expensive task. Typically for very large datasets, the dataset is split up into batches of examples sampled from the dataset, and each iteration of training is based on a batch, rather than the entire dataset. Fifth, apart from the modifications necessary to include the summation, this rule is identical to the LMS (also known as the Widrow-Hoff or delta) learning rule introduced in chapter 4, and the rule implements the same logic: if the output of the model is too large, then weights associated with positive inputs should be reduced; if the output is too small, then these weights should be increased. Moreover, the purpose and function of the learning rate hyperparameter (η) is the same as in the LMS rule: scale the weight adjustments to ensure that the adjustments aren't so large that the algorithm misses (or steps over) the best set of weights. Using this weight update rule, the gradient descent algorithm can be summarized as follows:

1. Construct a model using an initial set of weights.
2. Repeat until the model performance is good enough.
    a. Apply the current model to the examples in the dataset.
    b. Adjust each weight using the weight update rule.
3. Return the final model.

One consequence of the independent updating of weights, and the fact that weight updates are proportional to the local gradient on the associated error curve, is that the path the gradient descent algorithm follows to the lowest point on the error surface may not be a straight line. This is because the gradient of each of the component error curves may not be equal at each location on the error surface (the gradient for one of the weights may be steeper than the gradient for the other weight). As a result, one weight may be updated by a larger amount than another weight during a given iteration, and thus the descent to the valley floor may not follow a direct route. Figure 6.4 illustrates this phenomenon. Figure 6.4 presents a set of top-down views of a portion of a contour plot of an error surface. This error surface is a valley that is quite long

and narrow with steeper sides and gentler sloping ends; the steepness is reflected by the closeness of the contours. As a result, the search initially moves across the valley before turning toward the center of the valley. The plot on the left illustrates the first iteration of the gradient descent algorithm. The initial starting point is the location where the three arrows, in this plot, meet. The lengths of the dotted and dashed arrows represent the local gradients of the w_0 and w_1 error curves, respectively. The dashed arrow is longer than the dotted arrow reflecting the fact that the local gradient of the w_0 error curve is steeper than that of the w_1 error curve. In each iteration, each of the weights is updated in proportion to the gradient of their error curve; so in the first iteration, the update for w_0 is larger than for w_1 and therefore the overall movement is greater across the valley than along the valley. The thick black arrow illustrates the overall movement in the underlying weight space, resulting from the weight updates in this first iteration. Similarly, the middle plot illustrates the error gradients and overall weight update for the next iteration of gradient descent. The plot on the right shows the complete path of descent taken by the search process from initial location to the global minimum (the lowest point on the error surface).

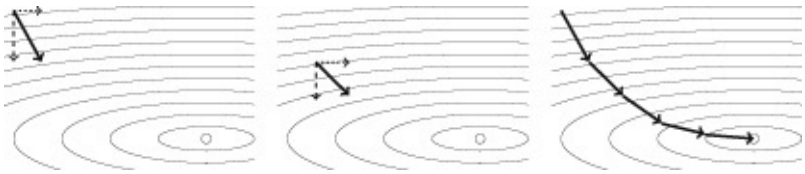


Figure 6.4 Top-down views of a portion of a contour plot of an error surface, illustrating the gradient descent path across the error surface. Each of the thick arrows illustrates the overall movement of the weight vector for a single iteration of the gradient descent algorithm. The length of dotted and dashed arrows represent the local gradient of the w_0 and w_1 error curves, respectively, for that iteration. The plot on the right shows the overall path taken to the global minimum of the error surface.

It is relatively straightforward to map the weight update rule over to training a single neuron. In this mapping, the weight w_0 is the bias term for a neuron, and the other weights are associated with the other inputs to the neuron. The derivation of the partial derivative of the SSE is dependent on the structure of the function that generates $\hat{y}$. The more complex this function is, the more complex the partial derivative becomes. The fact that the function a neuron defines includes both a weighted summation and an activation function means that the partial derivative of the SSE with respect to a weight in a neuron is more complex than the partial derivative given above. The inclusion of the activation function within the neuron results in an extra term in the partial derivative of the

SSE. This extra term is the derivative of the activation function with respect to the output from the weighted summation function. The derivative of the activation function is with respect to the output of the weighted summation function because this is the input that the activation function receives. The activation function does not receive the weight directly. Instead, the changes in the weight only affect the output of the activation function indirectly through the effect that these weight changes have on the output of the weighted summation. The main reason why the logistic function was such a popular activation function in neural networks for so long was that it has a very straightforward derivative with respect to its inputs. The gradient descent weight update rule for a neuron using the logistic function is as follows:

$$w_i^{t+1} = w_i^t + \left(\eta \times \underbrace{\sum_{j=1}^{n} \left((y_j^t - \hat{y}_j^t) \times \underbrace{(\hat{y}_j^t \times (1 - \hat{y}_j^t))}_{\substack{\text{derivative of the} \\ \text{logistic function} \\ \text{with respect to the} \\ \text{weighted summation}}} \times x_{j,i}^t \right)}_{\text{error gradient for } w_i} \right)$$

The fact that the weight update rule includes the derivative of the activation function means that the weight update rule will change if the activation function of the neuron is changed. However, this change will simply involve updating the derivative of the activation function; the overall structure of the rule will remain the same.

This extended weight update rule means that the gradient descent algorithm can be used to train a single neuron. It cannot, however, be used to train neural networks with multiple layers of neurons because the definition of the error gradient for a weight depends on the error of the output of the function, the term $y_j - \hat{y}_j$. Although it is possible to calculate the error of the output of a neuron in the output layer of the network by directly comparing the output with the expected output, it is not possible to calculate this error term directly for the neurons in the hidden layer of the network, and as a result it is not possible to calculate the error gradients for each weight. The backpropagation algorithm is a solution to the problem of calculating error gradients for the weights in the hidden layers of the network.

Training a Neural Network Using Backpropagation

The term backpropagation has two different meanings. The primary meaning is that it is an algorithm that can be used to calculate, for each neuron in a network, the sensitivity (gradient/rate-of-change) of the error of the network to changes in the weights. Once the error gradient for a weight has been calculated, the weight can then be adjusted to reduce the overall error of the network using a weight update rule similar to the gradient descent weight update rule. In this sense, the backpropagation algorithm is a solution to the credit assignment problem, introduced in chapter 4. The second meaning of backpropagation is that it is a complete algorithm for training a neural network. This second meaning encompasses the first sense, but also includes a learning rule that defines how the error gradients of the weights should be used to update the weights within the network. Consequently, the algorithm described by this second meaning involves a two-step process: solve the credit assignment problem, and then use the error gradients of the weights, calculated during credit assignment, to update the weights in the network. It is useful to distinguish between these two meanings of backpropagation because there are a number of different learning rules that can be used to update the weights, once the credit assignment problem has been resolved. The learning rule that is most commonly used with backpropagation is the gradient descent algorithm introduced earlier. The description of the backpropagation algorithm given here focuses on the first meaning of backpropagation, that of the algorithm being a solution to the credit assignment problem.

Backpropagation: The Two-Stage Algorithm

The backpropagation algorithm begins by initializing all the weights of the network using random values. Note that even a randomly initialized network can still generate an output when an input is presented to the network, although it is likely to be an output with a large error. Once the network weights have been initialized, the network can be trained by iteratively updating the weights so as to reduce the error of the network, where the error of the network is calculated in terms of the difference between the output generated by the network in response to an input pattern, and the expected output for that input, as defined in the training dataset. A crucial step in this iterative weight adjustment process involves solving the credit assignment problem, or, in other words, calculating

the error gradients for each weight in the network. The backpropagation algorithm solves this problem using a two-stage process. In first stage, known as the forward pass, an input pattern is presented to the network, and the resulting neuron activations flow forward through the network until an output is generated. Figure 6.5 illustrates the forward pass of the backpropagation algorithm. In this figure, the weighted summation of inputs calculated at each neuron (e.g., z_1 represents the weighted summation of inputs calculated for neuron 1) and the outputs (or activations, e.g., a_1 represents the activation for neuron 1) of each neuron is shown. The reason for listing the z_i and a_i values for each neuron in this figure is to highlight the fact that during the forward pass both of these values, for each neuron, are stored in memory. The reason they are stored in memory is that they are used in the backward pass of the algorithm. The z_i value for a neuron is used to calculate the update to the weights on input connections to the neuron. The a_i value for a neuron is used to calculate the update to the weights on the output connections from a neuron. The specifics of how these values are used in the backward pass will be described below.

The second stage, known as the backward pass, begins by calculating an error gradient for each neuron in the output layer. These error gradients represent the sensitivity of the network error to changes in the weighted summation calculation of the neuron, and they are often denoted by the shorthand notation δ (pronounced delta) with a subscript indicating the neuron. For example, δ_k is the gradient of the network error with respect to small changes in the weighted summation calculation of the neuron *k*. It is important to recognize that there are two different error gradients calculated in the backpropagation algorithm:

1. The first is the δ value for each neuron. The δ for each neuron is the rate of change of the error of the network with respect to changes in the weighted summation calculation of the neuron. There is one δ for each neuron. It is these δ error gradients that the algorithm backpropagates.
2. The second is the error gradient of the network with respect to changes in the weights of the network. There is one of these error gradients for each weight in the network. These are the error gradients that are used to update the weights in the network. However, it is necessary to first calculate the δ term for each neuron (using backpropagation) in order to calculate the error gradients for the weights.

Note there is only a single δ per neuron, but there may be many weights associated with that neuron, so the δ term for a neuron may be used in the calculation of multiple weight error gradients.

Once the δs for the output neurons have been calculated, the δs for the neurons in the last hidden layer are then calculated. This is done by assigning a portion of the δ from each output neuron to each hidden neuron that is directly connected to it. This assignment of blame, from output neuron to hidden neuron, is dependent on the weight of the connection between the neurons, and the activation of the hidden neuron during the forward pass (this is why the activations are recorded in memory during the forward pass). Once the blame assignment, from the output layer, has been completed, the δ for each neuron in the last hidden layer is calculated by summing the portions of the δs assigned to the neuron from all of the output neurons it connects to. The same process of blame assignment and summing is then repeated to propagate the error gradient back from the last layer of hidden neurons to the neurons in the second last layer, and so on, back to the input layer. It is this backward propagation of δs through the network that gives the algorithm its name. At the end of this backward pass there is a δ calculated for each neuron in the network (i.e., the credit assignment problem has been solved) and these δs can then be used to update the weights in the network (using, for example, the gradient descent algorithm introduced earlier). Figure 6.6 illustrates the backward pass of the backpropagation algorithm. In this figure, the δs get smaller and smaller as the backpropagation process gets further from the output layer. This reflects the vanishing gradient problem discussed in chapter 4 that slows down the learning rate of the early layers of the network.

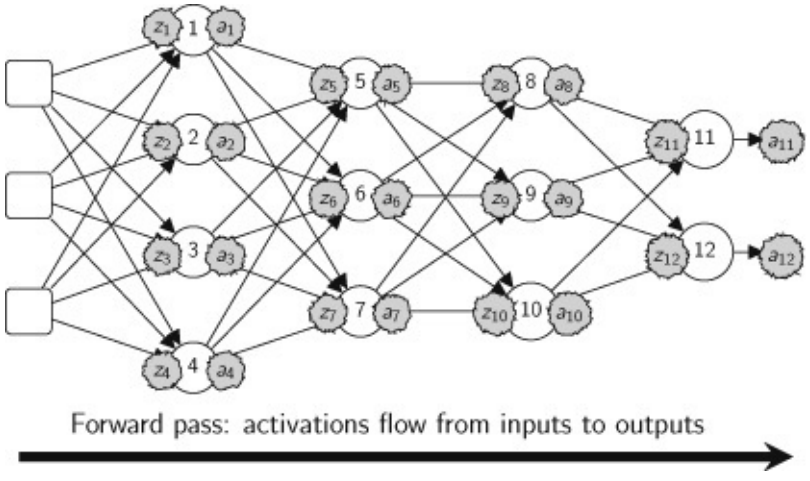


Figure 6.5 The forward pass of the backpropagation algorithm.

In summary, the main steps within each iteration of the backpropagation algorithm are as follows:

1. Present an input to the network and allow the neuron activations to flow forward through the network until an output is generated. Record both the weighted sum and the activation of each neuron.

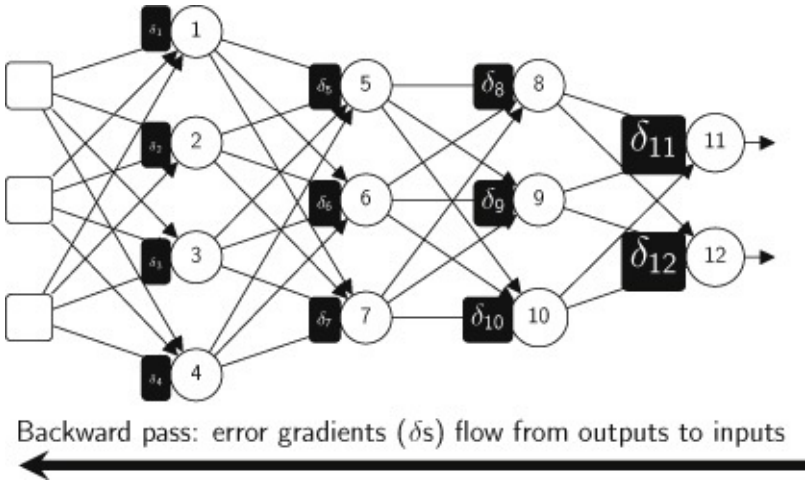


Figure 6.6 The backward pass of the backpropagation algorithm.

2. Calculate a δ (delta) error gradient for each neuron in the output layer.
3. Backpropagate the δ error gradients to obtain a δ (delta) error gradient for each neuron in the network.
4. Use the δ error gradients and a weight update algorithm, such as gradient descent, to calculate the error gradients for the weights and use these to update the weights in the network.

The algorithm continues iterating through these steps until the error of the network is reduced (or converged) to an acceptable level.

Backpropagation: Backpropagating the δ s

A δ term of a neuron describes the error gradient for the network with respect to changes in the weighted summation of inputs calculated by the neuron. To help make this more concrete, figure 6.7 (top) breaks open the processing stages within a neuron k and uses the term z_k to denote the result of the weighted summation within the neuron. The neuron in this figure receives inputs (or activations) from three other neurons (h, i, j), and z_k is the weighted sum of these

activations. The output of the neuron, a_k, is then calculated by passing z_k through a nonlinear activation function, φ, such as the logistic function. Using this notation a δ for a neuron k is the rate of change of the error of the network with respect to small changes in the value of z_k. Mathematically, this term is the partial derivative of the networks error with respect to z_k:

$$\delta_k = \frac{\partial \text{Error}}{\partial z_k}$$

No matter where in a network a neuron is located (output layer or hidden layer), the δ for the neuron is calculated as the product of two terms:

1. the rate of change of the network error in response to changes in the neuron's activation (output): $\partial E / \partial a_k$;

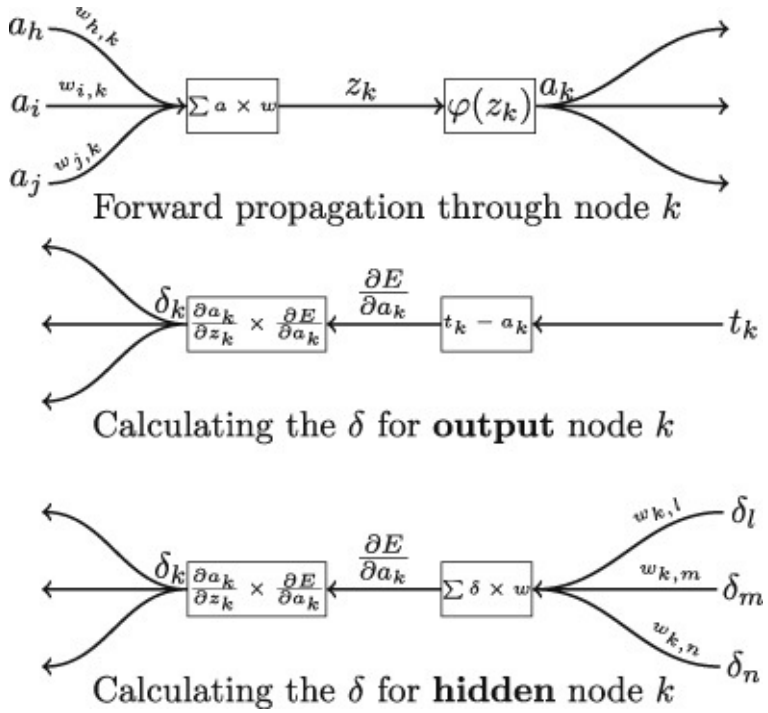


Figure 6.7 Top: the forward propagation of activations through the weighted sum and activation function of a neuron. Middle: The calculation of the δ term for an output neuron (t_k is the expected activation for the neuron and a_k is the actual activation). Bottom: The calculation of the δ term for a hidden neuron. This figure is loosely inspired by figure 5.2 and figure 5.3 in Reed and Marks II 1999.

2. the rate of change of the activation of the neuron with respect to

changes in the weighted sum of inputs to the neuron: $\partial a_k / \partial z_k$.

$$\delta_k = \frac{\partial E}{\partial a_k} \times \frac{\partial a_k}{\partial z_k}$$

Figure 6.7 (middle) illustrates how this product is calculated for neurons in the output layer of a network. The first step is to calculate the rate of change of the error of the network with respect to the output of the neuron, the term $\partial E / \partial a_k$. Intuitively, the larger the difference between the activation of a neuron, a_k, and the expected activation, t_k, the faster the error can be changed by changing the activation of the neuron. So the rate of change of the error of the network with respect to changes in the activation of an output neuron k can be calculated by subtracting the neuron's activation (a_k) from the expected activation (t_k):

$$\frac{\partial E}{\partial a_k} = t_k - a_k$$

This term connects the error of the network to the output of the neuron. The neuron's δ, however, is the rate of change of the error with respect to the input to the activation function (z_k), not the output of that function (a_k). Consequently, in order to calculate the δ for the neuron, the $\partial E / \partial a_k$ value must be propagated back through the activation function to connect it to the input to the activation function. This is done by multiplying $\partial E / \partial a_k$ by the rate of change of the activation function with respect to the input value to the function, z_k. In figure 6.7, the rate of change of the activation function with respect to its input is denoted by the term: $\partial a_k / \partial z_k$. This term is calculated by plugging the value z_k (stored from the forward pass through the network) into the equation of the derivative of the activation function with respect to z_k. For example, the derivative of the logistic function with respect to its input is:

$$\frac{\partial \text{logistic}(z)}{\partial z} = \text{logistic}(z) \times (1 - \text{logistic}(z))$$

Figure 6.8 plots this function and shows that plugging a z_k value into this equation will result in a value between 0 and 0.25. For example, figure 6.8

shows that if $z_k = 0$ then $\partial a_k / \partial z_k = 0.25$. This is why the weighted summation value for each neuron (z_k) is stored during the forward pass of the algorithm.

The fact[1] that the calculation of a neuron's δ involves a product that includes the derivative of the neuron's activation function makes it necessary to be able to take the derivative of the neuron's activation function. It is not possible to take the derivative of a threshold activation function because there is a discontinuity in the function at the threshold. As a result, the backpropagation algorithm does not work for networks composed of neurons that use threshold activation functions. This is one of the reasons why neural networks moved away from threshold activation and started to use the logistic and tanh activation functions. The logistic and tanh functions both have very simple derivatives and this made them particularly suitable to backpropagation.

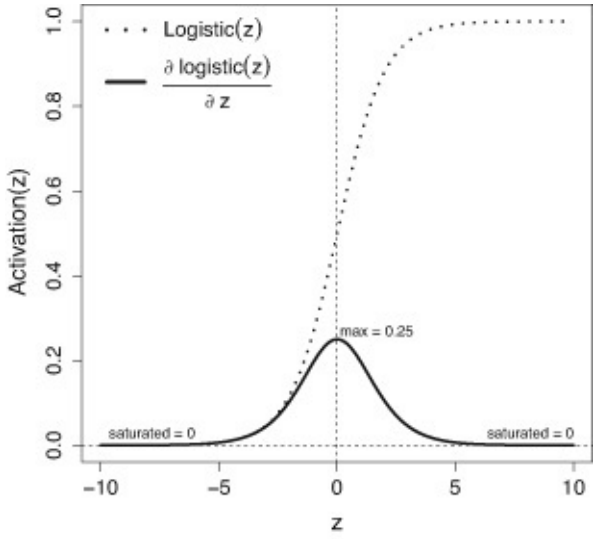


Figure 6.8 Plots of the logistic function and the derivative of the logistic function.

Figure 6.7 (bottom) illustrates how the δ for a neuron in a hidden layer is calculated. This involves the same product of terms as was used for neurons in the output layer. The difference is that the calculation of the $\partial E / \partial a_k$ is more complex for hidden units. For hidden neurons, it is not possible to directly connect the output of the neuron with the error of a network. The output of a hidden neuron only indirectly affects the overall error of the network through the variations that it causes in the downstream neurons that receive the output as input, and the magnitude of these variations is dependent on the weight each of these downstream neurons applies to the output. Furthermore, this indirect effect on the network error is in turn dependent on the sensitivity of the network error

to these later neurons, that is, their δ values. Consequently, the sensitivity of the network error to the output of a hidden neuron can be calculated as a weighted sum of the δ values of the neurons immediately downstream of the neuron:

$$\frac{\partial E}{\partial a_k} = \sum_{i=1}^{N} w_{k,i} \times \delta_i$$

As a result, the error terms (the δ values) for all the downstream neurons to which a neuron's output is passed in the forward pass must be calculated before the $\partial E / \partial a_k$ for neuron *k* can be calculated. This, however, is not a problem because in the backward pass the algorithm is working backward through the network and will have calculated the δ terms for the downstream neurons before it reaches neuron *k*.

For hidden neurons, the other term in the δ product, $\partial a_k / \partial z_k$, is calculated in the same way as it is calculated for output neurons: the z_k value for the neuron (the weighted summation of inputs, stored during the forward pass through the network) is plugged into the derivative of the neuron's activation function with respect to z_k.

Backpropagation: Updating the Weights

The fundamental principle of the backpropagation algorithm in adjusting the weights in a network is that each weight in a network should be updated in proportion to the sensitivity of the overall error of the network to changes in that weight. The intuition is that if the overall error of the network is not affected by a change in a weight, then the error of the network is independent of that weight, and, therefore, the weight did not contribute to the error. The sensitivity of the network error to a change in an individual weight is measured in terms of the rate of change of the network error in response to changes in that weight.

The fundamental principle of the backpropagation algorithm in adjusting the weights in a network is that each weight in a network should be updated in proportion to the sensitivity of the overall error of the network to changes in that weight.

The overall error of a network is a function with multiple inputs: both the

inputs to the network and all the weights in the network. So, the rate of change of the error of a network in response to changes in a given network weight is calculated by taking the partial derivative of the network error with respect to that weight. In the backpropagation algorithm, the partial derivative of the network error for a given weight is calculated using the chain rule. Using the chain rule, the partial derivative of the network error with respect a weight $w_{j,k}$ on the connection between a neuron j and a neuron k is calculated as the product of two terms:

1. the first term describes the rate of change of the weighted sum of inputs in neuron k with respect to changes in the weight $\partial z_k / \partial w_{j,k}$;
2. and the second term describes the rate of change of the network error in response to changes in the weighted sum of inputs calculated by the neuron k. (This second term is the δ_k for neuron k.)

Figure 6.9 shows how the product of these two terms connects a weight to the output error of the network. The figure shows the processing of the last two neurons (k and l) in a network with a single path of activation. Neuron k receives a single input a_j and the output from neuron k is the sole input to neuron l. The output of neuron l is the output of the network. There are two weights in this portion of the network, $w_{j,k}$ and $w_{k,l}$.

The calculations shown in figure 6.9 appear complicated because they contain a number of different components. However, as we will see, by stepping through these calculations, each of the individual elements is actually easy to calculate; it's just keeping track of all the different elements that poses a difficulty.

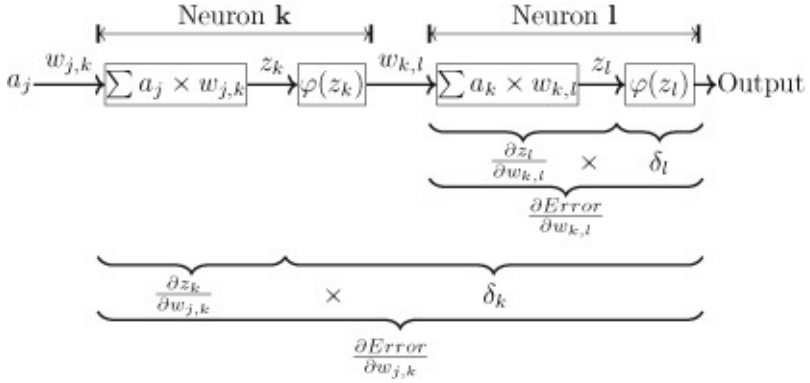


Figure 6.9 An illustration of how the product of derivatives connects weights in the network to the error of the network.

Focusing on $w_{k,l}$, this weight is applied to an input of the output neuron of the network. There are two stages of processing between this weight and the network output (and error): the first is the weighted sum calculated in neuron l; the second is the nonlinear function applied to this weighted sum by the activation function of neuron l. Working backward from the output, the δ_l term is calculated using the calculation shown in the middle figure of figure 6.7: the difference between the target activation for the neuron and the actual activation is calculated and is multiplied by the partial derivative of the neuron's activation function with respect to its input (the weighted sum z_k), $\partial a_l / \partial z_l$. Assuming that the activation function used by neuron l is the logistic function, the term $\partial a_l / \partial z_l$ is calculated by plugging in the value z_l (stored during the forward pass of the algorithm) into the derivation of the logistic function:

$$\frac{\partial a_l}{\partial z_l} = \frac{\partial \text{logistic}(z_l)}{\partial z_l} = \text{logistic}(z_l) \times (1 - \text{logistic}(z_l))$$

So the calculation of δ_l under the assumption that neuron l uses a logistic function is:

$$\delta_l = \text{logistic}(z_l) \times (1 - \text{logistic}(z_l)) \times (t_l - a_l)$$

The δ_l term connects the error of the network to the input to the activation function (the weighted sum z_l). However, we wish to connect the error of the network back to the weight $w_{k,l}$. This is done by multiplying the δ_l term by the partial derivative of the weighted summation function with respect to weight $w_{k,l}$: $\partial z_l / \partial w_{k,l}$. This partial derivative describes how the output of the weighted sum function z_l changes as the weight $w_{k,l}$ changes. The fact that the weighted summation function is a linear function of weights and activations means that in the partial derivative with respect to a particular weight all the terms in the function that do not involve the specific weight go to zero (are considered constants) and the partial derivative simplifies to just the input associated with that weight, in this instance input a_k.

$$\frac{\partial z_l}{\partial w_{k,l}} = a_k$$

This is why the activations for each neuron in the network are stored in the forward pass. Taken together these two terms, $\partial z_l / \partial w_{k,l}$ and δ_l, connect the weight $w_{k,l}$ to the network error by first connecting the weight to z_l, and then connecting z_l to the activation of the neuron, and thereby to the network error. So, the error gradient of the network with respect to changes in weight $w_{k,l}$ is calculated as:

$$\frac{\partial \text{Error}}{\partial w_{k,l}} = \frac{\partial z_l}{\partial w_{k,l}} \times \delta_l = a_k \times \delta_l$$

The other weight in the figure 6.9 network, $w_{k,l}$, is deeper in the network, and, consequently, there are more processing steps between it and the network output (and error). The δ term for neuron k is calculated, through backpropagation (as shown at the bottom of figure 6.7), using the following product of terms:

$$\delta_k = \frac{\partial a_k}{\partial z_k} \times (w_{k,l} \times \delta_l)$$

Assuming the activation function used by neuron k is the logistic function, then the term $\partial a_k / \partial z_k$ is calculated in a similar way to $\partial a_l / \partial z_l$: the value z_k is plugged into the equation for the derivative of the logistic function. So, written out in long form the calculation of δ_k is:

$$\delta_k = \text{logistic}(z_k) \times (1 - \text{logistic}(z_k)) \times (w_{k,l} \times \delta_l)$$

However, in order to connect the weight $w_{j,k}$ with the error of the network, the term δ_k must be multiplied by the partial derivative of the weighted summation function with respect to the weight: $\partial z_k / \partial w_{j,k}$. As described above, the partial derivative of a weighted sum function with respect to a weight reduces to the input associated with the weight $w_{j,k}$ (i.e., a_j); and the gradient of the networks

error with respect to the hidden weight $w_{j,k}$ is calculated by multiplying a_j by δ_k. Consequently, the product of the terms ($\partial z_k / \partial w_{j,k}$ and δ_k) forms a chain connecting the weight $w_{j,k}$ to the network error. For completeness, the product of terms for $w_{j,k}$, assuming logistic activation functions in the neurons, is:

$$\frac{\partial \text{Error}}{\partial w_{j,k}} = \frac{\partial z_k}{\partial w_{j,k}} \times \delta_k = a_j \times \delta_k$$

Although this discussion has been framed in the context of a very simple network with only a single path of connections, it generalizes to more complex networks because the calculation of the δ terms for hidden units already considers the multiple connections emanating from a neuron. Once the gradient of the network error with respect to a weight has been calculated ($\partial \text{Error} / w_{j,k} = \delta_k \times a_j$), the weight can be adjusted so as to reduce the weight of the network using the gradient descent weight update rule. Here is the weight update rule, specified using the notation from backpropagation, for the weight on the connection between neuron j and neuron k during iteration t of the algorithm:

$$w_{j,k}^{t+1} = w_{j,k}^{t} + (\eta \times \delta_k \times a_j)$$

Finally, an important caveat on training neural networks with backpropagation and gradient descent is that the error surface of a neural network is much more complex than that of a linear models. Figure 6.3 illustrated the error surface of a linear model as a smooth convex bowl with a single global minimum (a single best set of weights). However, the error surface of a neural network is more like a mountain range with multiple valleys and peaks. This is because each of the neurons in a network includes a nonlinear function in its mapping of inputs to outputs, and so the function implemented by the network is a nonlinear function. Including a nonlinearity within the neurons of a network increases the expressive power of the network in terms of its ability to learn more complex functions. However, the price paid for this is that the error surface becomes more complex and the gradient descent algorithm is no longer guaranteed to find the set of weights that define the global minimum on the error surface; instead it may get stuck within a minima (local minimum). Fortunately, however, backpropagation and gradient descent can still often find sets of weights that define useful

models, although searching for useful models may require running the training process multiple times to explore different parts of the error surface landscape.

7

The Future of Deep Learning

On March 27, 2019, Yoshua Bengio, Geoffrey Hinton, and Yann LeCun jointly received the ACM A.M. Turing award. The award recognized the contributions they have made to deep learning becoming the key technology driving the modern artificial intelligence revolution. Often described as the “Nobel Prize for Computing,” the ACM A.M Turing award carries a $1 million prize. Sometimes working together, and at other times working independently or in collaboration with others, these three researchers have, over a number of decades of work, made numerous contributions to deep learning, ranging from the popularization of backpropagation in the 1980s, to the development of convolutional neural networks, word embeddings, attention mechanisms in networks, and generative adversarial networks (to list just some examples). The announcement of the award noted the astonishing recent breakthroughs that deep learning has led to in computer vision, robotics, speech recognition, and natural language processing, as well as the profound impact that these technologies are having on society, with billions of people now using deep learning based artificial intelligence on a daily basis through smart phones applications. The announcement also highlighted how deep learning has provided scientists with powerful new tools that are resulting in scientific breakthroughs in areas as diverse as medicine and astronomy. The awarding of this prize to these researchers reflects the importance of deep learning to modern science and society. The transformative effects of deep learning on technology is set to increase over the coming decades with the development and adoption of deep learning continuing to be driven by the virtuous cycle of ever larger datasets, the development of new algorithms, and improved hardware. These trends are not stopping, and how the deep learning community responds to them will drive growth and innovations within the field over the coming years.

Big Data Driving Algorithmic Innovations

Chapter 1 introduced the different types of machine learning: supervised, unsupervised, and reinforcement learning. Most of this book has focused on supervised learning, primarily because it is the most popular form of machine learning. However, a difficulty with supervised learning is that it can cost a lot of money and time to annotate the dataset with the necessary target labels. As datasets continue to grow, the data annotation cost is becoming a barrier to the development of new applications. The ImageNet dataset[1] provides a useful example of the scale of the annotation task involved in deep learning projects. This data was released in 2010, and is the basis for the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). This is the challenge that the AlexNet CNN won in 2012 and the ResNet system won in 2015. As was discussed in chapter 4, AlexNet winning the 2012 ILSVRC challenge generated a lot of excitement about deep learning models. However, the AlexNet win would not have been possible without the creation of the ImageNet dataset. This dataset contains more than fourteen million images that have been manually annotated to indicate which objects are present in each image; and more than one million of the images have actually been annotated with the bounding boxes of the objects in the image. Annotating data at this scale required a significant research effort and budget, and was achieved using crowdsourcing platforms. It is not feasible to create annotated datasets of this size for every application.

As datasets continue to grow, the data annotation cost is becoming a barrier to the development of new applications.

One response to this annotation challenge has been a growing interest in unsupervised learning. The autoencoder models used in Hinton's pretraining (see chapter 4) are one neural network approach to unsupervised learning, and in recent years different types of autoencoders have been proposed. Another approach to this problem is to train generative models. Generative models attempt to learn the distribution of the data (or, to model the process that generated the data). Similar to autoencoders, generative models are often used to learn a useful representation of the data prior to training a supervised model. Generative adversarial networks (GANs) are an approach to training generative models that has received a lot of attention in recent years (Goodfellow et al.

2014). A GAN consists of two neural networks, a generative model and a discriminative model, and a sample of real data. The models are trained in an adversarial manner. The task of the discriminative model is to learn to discriminate between real data sampled from the dataset, and fake data that has been synthesized by the generator. The task of the generator is to learn to synthesize fake data that can fool the discriminative model. Generative models trained using a GAN can learn to synthesize fake images that mimic an artistic style (Elgammal et al. 2017), and also to synthesize medical images along with lesion annotations (Frid-Adar et al. 2018). Learning to synthesize medical images, along with the segmentation of the lesions in the synthesized image, opens the possibility of automatically generating massive labeled datasets that can be used for supervised learning. A more worrying application of GANs is the use of these networks to generate *deep fakes*: a deep fake is a fake video of a person doing something they never did that is created by swapping their face into a video of someone else. Deep fakes are very hard to detect, and have been used maliciously on a number of occasions to embarrass public figures, or to spread fake news stories.

Another solution to the data labeling bottleneck is that rather than training a new model from scratch for each new application, we rather repurpose models that have been trained on a similar task. Transfer learning is the machine learning challenge of using information (or representations) learned on one task to aid learning on another task. For transfer learning to work, the two tasks should be from related domains. Image processing is an example of a domain where transfer learning is often used to speed up the training of models across different tasks. Transfer learning is appropriate for image processing tasks because low-level visual features, such as edges, are relatively stable and useful across nearly all visual categories. Furthermore, the fact that CNN models learn a hierarchy of visual feature, with the early layers in CNN learning functions that detect these low-level visual features in the input, makes it possible to repurpose the early layers of pretrained CNNs across multiple image processing projects. For example, imagine a scenario where a project requires an image classification model that can identify objects from specialized categories for which there are no samples in general image datasets, such as ImageNet. Rather than training a new CNN model from scratch, it is now relatively standard to first download a state-of-the-art model (such as the Microsoft ResNet model) that has been trained on ImageNet, then replace the later layers of the model with a new set of layers, and finally to train this new hybrid-model on a relatively small dataset that has been labeled with the appropriate categories for the project. The later

layers of the state-of-the-art (general) model are replaced because these layers contain the functions that combine the low-level features into the task specific categories the model was originally trained to identify. The fact that the early layers of the model have already been trained to identify the low-level visual features speeds up the training and reduces the amount of data needed to train the new project specific model.

The increased interest in unsupervised learning, generative models, and transfer learning can all be understood as a response to the challenge of annotating increasingly large datasets.

The Emergence of New Models

The rate of emergence of new deep learning models is accelerating every year. A recent example is capsule networks (Hinton et al. 2018; Sabour et al. 2017). Capsule networks are designed to address some of the limitations of CNNs. One problem with CNNs, sometimes known as the Picasso problem, is the fact that a CNN ignores the precise spatial relationships between high-level components within an object's structure. What this means in practice is that a CNN that has been trained to identify faces may learn to identify the shapes of eyes, the nose, and the mouth, but will not learn the required spatial relationships between these parts. Consequently, the network can be fooled by an image that contains these body parts, even if they are not in the correct relative position to each other. This problem arises because of the pooling layers in CNNs that discard positional information.

At the core of capsule networks is the intuition that the human brain learns to identify object types in a viewpoint invariant manner. Essentially, for each object type there is an object class that has a number of instantiation parameters. The object class encodes information such as the relative relationship of different object parts to each other. The instantiation parameters control how the abstract description of an object type can be mapped to the specific instance of the object that is currently in view (for example, its pose, scale, etc.).

A capsule is a set of neurons that learns to identify whether a specific type of object or object part is present at a particular location in an image. A capsule outputs an activity vector that represents the instantiation parameters of the object instance, if one is present at the relevant location. Capsules are embedded within convolutional layers. However, capsule networks replace the pooling process, which often defines the interface between convolutional layers, with a process called dynamic routing. The idea behind dynamic routing is that each capsule in one layer in the network learns to predict which capsule in the next layer is the most relevant capsule for it to forward its output vector to.

At the time or writing, capsule networks have the state-of-the-art performance on the MNIST handwritten digit recognition dataset that the original CNNs were trained on. However, by today's standards, this is a relatively small dataset, and capsule networks have not been scaled to larger datasets. This is partly because the dynamic routing process slows down the training of capsule networks. However, if capsule networks are successfully scaled, then they may introduce

an important new form of model that extends the ability of neural networks to analyze images in a manner much closer to the way humans do.

Another recent model that has garnered a lot of interest is the transformer model (Vaswani et al. 2017). The transformer model is an example of a growing trend in deep learning where models are designed to have sophisticated internal attention mechanisms that enable a model to dynamically select subsets of the input to focus on when generating an output. The transformer model has achieved state-of-the-art performance on machine translation for some language pairs, and in the future this architecture may replace the encoder-decoder architecture described in chapter 5. The BERT (Bidirectional Encoder Representations from Transformers) model has built on the Transformer architecture (Devlin et al. 2018). The BERT development is particularly interesting because at its core is the idea of transfer learning (as discussed above in relation to the data annotation bottleneck). The basic approach to creating a natural language processing model with BERT is to pretrain a model for a given language using a large unlabeled dataset (the fact that the dataset is unlabeled means that it is relatively cheap to create). This pretrained model can then be used as the basis to create a models for specific tasks for the language (such as sentiment classification or question answering) by fine-tuning the pretrained model using supervised learning and a relatively small annotated dataset. The success of BERT has shown this approach to be tractable and effective in developing state-of-the-art natural language processing systems.

New Forms of Hardware

Today's deep learning is powered by graphics processing units (GPUs): specialized hardware that is optimized to do fast matrix multiplications. The adoption, in the late 2000s, of commodity GPUs to speed up neural network training was a key factor in many of the breakthroughs that built momentum behind deep learning. In the last ten years, hardware manufacturers have recognized the importance of the deep learning market and have developed and released hardware specifically designed for deep learning, and which supports deep learning libraries, such as TensorFlow and PyTorch. As datasets and networks continue to grow in size, the demand for faster hardware continues. At the same time, however, there is a growing recognition of the energy costs associated with deep learning, and people are beginning to look for hardware solutions that have a reduced energy footprint.

Neuromorphic computing emerged in the late 1980s from the work of Carver Mead.[2] A neuromorphic chip is composed of a very-large-scale integrated (VLSI) circuit, connecting potentially millions of low-power units known as spiking neurons. Compared with the artificial neurons used in standard deep learning systems, the design of a spiking neuron is closer to the behavior of biological neurons. In particular, a spiking neuron does not fire in response to the set of input activations propagated to it at a particular time point. Instead, a spiking neuron maintains an internal state (or activation potential) that changes through time as it receives activation pulses. The activation potential increases when new activations are received, and decays through time in the absence of incoming activations. The neuron fires when its activation potential surpasses a specific threshold. Due to the temporal decay of the neuron's activation potential, a spiking neuron only fires if it receives the requisite number of input activations within a time window (a spiking pattern). One advantage of this temporal based processing is that spiking neurons do not fire on every propagation cycle, and this reduces the amount of energy the network consumes.

In comparison with traditional CPU design, neuromorphic chips have a number of distinctive characteristics, including:

1. Basic building blocks: traditional CPUs are built using transistor based logic gates (e.g., AND, OR, NAND gates), whereas neuromorphic chips are built using spiking neurons.
2. Neuromorphic chips have an analog aspect to them: in a traditional

digital computer, information is sent in high-low electrical bursts in sync with a central clock; in a neuromorphic chip, information is sent as patterns of high-low signals that vary through time.

3. Architecture: the architecture of traditional CPUs is based on the von Neumann architecture, which is intrinsically centralized with all the information passing through the CPU. A neuromorphic chip is designed to allow massive parallelism of information flow between the spiking neurons. Spiking neurons communicate directly with each other rather than via a central information processing hub.
4. Information representation is distributed through time: the information signals propagated through a neuromorphic chip use a distributed representation, similar to the distributed representations discussed in chapter 4, with the distinction that in a neuromorphic chip these representations are also distributed through time. Distributed representations are more robust to information loss than local representations, and this is a useful property when passing information between hundreds of thousands, or millions, of components, some of which are likely to fail.

Currently there are a number of major research projects focused on neuromorphic computing. For example, in 2013 the European Commission allocated one billion euros in funding to the ten-year Human Brain Project.[3] This project directly employs more than five hundred scientists, and involves research from more than a hundred research centers across Europe. One of the projects key objectives is the development of neuromorphic computing platforms capable of running a simulation of a complete human brain. A number of commercial neuromorphic chips have also been developed. In 2014, IBM launched the TrueNorth chip, which contained just over a million neurons that are connected together by over 286 million synapses. This chip uses approximately 1/10,000th the power of a conventional microprocessor. In 2018, Intel Labs announced the Loihi (pronounced *low-ee-hee*) neuromorphic chip. The Loihi chip has 131,072 neurons connected together by 130,000,000 synapses. Neuromorphic computing has the potential to revolutionize deep learning; however, it still faces a number of challenges, not least of which is the challenge of developing the algorithms and software patterns for programming this scale of massively parallel hardware.

Finally, on a slightly longer time horizon, quantum computing is another stream of hardware research that has the potential to revolutionize deep learning. Quantum computing chips are already in existence; for example, Intel has created a 49-qubit quantum test chip, code named Tangle Lake. A qubit is the

quantum equivalent of a binary digit (bit) in traditional computing. A qubit can store more than one bit of information; however, it is estimated that it will require a system with one million or more qubits before quantum computing will be useful for commercial purposes. The current time estimate for scaling quantum chips to this level is around seven years.

The Challenge of Interpretability

Machine learning, and deep learning, are fundamentally about making data-driven decisions. Although deep learning provides a powerful set of algorithms and techniques to train models that can compete (and in some cases outperform) humans on a range of decision-making tasks, there are many situations where a decision by itself is not sufficient. Frequently, it is necessary to provide not only a decision but also the reasoning behind a decision. This is particularly true when the decision affects a person, be it a medical diagnosis or a credit assessment. This concern is reflected in privacy and ethics regulations in relation to the use of personal data and algorithmic decision-making pertaining to individuals. For example, Recital 71[4] of the General Data Protection Regulations (GDPR) states that individuals, affected by a decision made by an automated decision-making process, have the right to an explanation with regards to how the decision was reached.

Different machine learning models provide different levels of interpretability with regard to how they reach a specific decision. Deep learning models, however, are possibly the least interpretable. At one level of description, a deep learning model is quite simple: it is composed of simple processing units (neurons) that are connected together into a network. However, the scale of the networks (in terms of the number of neurons and the connections between them), the distributed nature of the representations, and the successive transformations of the input data as the information flows deeper into the network, makes it incredibly difficult to interpret, understand, and therefore explain, how the network is using an input to make a decision.

The legal status of the right to explanation within GDPR is currently vague, and the specific implications of it for machine learning and deep learning will need to be worked out in the courts. This example does, however, highlight the societal need for a better understanding of how deep learning models use data. The ability to interpret and understand the inner workings of a deep learning model is also important from a technical perspective. For example, understanding how a model uses data can reveal if a model has an unwanted bias in how it makes its decisions, and also reveal the corner cases that the model will fail on. The deep learning and the broader artificial intelligence research communities are already responding to this challenge. Currently, there are a number of projects and conferences focused on topics such as explainable artificial intelligence, and human interpretability in machine learning.

Chis Olah and his colleagues summarize the main techniques currently used to examine the inner workings of deep learning models as: feature visualization, attribution, and dimensionality reduction (Olah et al. 2018). One way to understand how a network processes information is to understand what inputs trigger particular behaviors in a network, such as a neuron firing. Understanding the specific inputs that trigger the activation of a neuron enables us to understand what the neuron has learned to detect in the input. The goal of feature visualization is to generate and visualize inputs that cause a specific activity within a network. It turns out that optimization techniques, such a backpropogation, can be used to generate these inputs. The process starts with a random generated input and the input is then iteratively updated until the target behavior is triggered. Once the required necessary input has been isolated, it can then be visualized in order to provide a better understanding of what the network is detecting in the input when it responds in a particular way. Attribution focuses on explaining the relationship between neurons, for example, how the output of a neuron in one layer of the network contributes to the overall output of the network. This can be done by generating a saliency (or heat-map) for the neurons in a network that captures how much weight the network puts on the output of a neuron when making a particular decision. Finally, much of the activity within a deep learning network is based on the processing of high-dimensional vectors. Visualizing data enables us to use our powerful visual cortex to interpret the data and the relationships within the data. However, it is very difficult to visualize data that has a dimensionality greater than three. Consequently, visualization techniques that are able to systematically reduce the dimensionality of high-dimensional data and visualize the results are incredibly useful tools for interpreting the flow of information within a deep network. t-SNE[5] is a well-known technique that visualizes high-dimensional data by projecting each datapoint into a two- or three-dimensional map (van der Maaten and Hinton 2008). Research on interpreting deep learning networks is still in its infancy, but in the coming years, for both societal and technical reasons, this research is likely to become a more central concern to the broader deep learning community.

Final Thoughts

Deep learning is ideally suited for applications involving large datasets of high-dimensional data. Consequently, deep learning is likely to make a significant contribution to some of the major scientific challenges of our age. In the last two decades, breakthroughs in biological sequencing technology have made it possible to generate high-precision DNA sequences. This genetic data has the potential to be the foundation for the next generation of personalized precision medicine. At the same time, international research projects, such as the Large Hadron Collider and Earth orbit telescopes, generate huge amounts of data on a daily basis. Analyzing this data can help us to understand the physics of our universe at the smallest and the biggest scales. In response to this flood of data, scientists are, in ever increasing numbers, turning to machine learning and deep learning to enable them to analyze this data.

One way to understand how a network processes information is to understand what inputs trigger particular behaviors in a network, such as a neuron firing.

At a more mundane level, however, deep learning already directly affects our lives. It is likely, that for the last few years, you have unknowingly been using deep learning models on a daily basis. A deep learning model is probably being invoked every time you use an internet search engine, a machine translation system, a face recognition system on your camera or social media website, or use a speech interface to a smart device. What is potentially more worrying is that the trail of data and metadata that you leave as you move through the online world is also being processed and analzsed using deep learning models. This is why it is so important to understand what deep learning is, how it works, what is it capable of, and its current limitations.

Glossary

ctivation Function

function that takes as input the result of the weighted sum of the inputs to a neuron and applies a nonlinear mapping to this weighted sum. Including an activation function within the neurons of a network enables the network to learn a nonlinear mapping. Examples of commonly used activation functions include: logistic, tanh, and ReLU.

rtificial Intelligence

ıe field of research that is focused on developing computational systems that can perform tasks and activities normally considered to require human intelligence.

ıckpropagation

ıckpropagation is an algorithm used to train a neural network with hidden layers of neurons. During training, the weights in a network are iteratively updated to reduce the error of the network. In order to update the weights on the links coming into a specific neuron in a network, it is necessary to first calculate an estimate of the contribution of the output of that neuron to the overall error of the network. The backpropagation algorithm is a solution to calculating these estimates for each neuron in the network. Once these errors estimates have been calculated for each neuron, the weights of the neurons can be updated using an optimization algorithm such as gradient descent. Backpropagation works in two phases: a forward pass and a backward pass. In the forward pass, an example is presented to the network and the overall error of the network is calculated at the output layer of the network by comparing the output of the network with the expected output for the example specified in the dataset. In the backward pass, the error of the network is shared back through the network with each neuron receiving a portion of blame for the error in proportion to the sensitivity of the error to changes in the output of that neuron. The process of sharing back the errors through the network is known as backpropagating the errors and this is where the algorithm gets its name.

ɔnvolutional Neural Network

convolutional neural network is a network that has at least one convolutional layer in it. A convolution layer is composed of a set of neurons that share the same set of weights and whose combined receptive fields cover an entire input. The union of the outputs of such a set of neurons is known as a feature map. In many convolutional neural networks, features maps are passed through a ReLU activation layer and then a pooling layer.

ıtaset

collection of instances with each instances described in terms of a set of features. In its most basic form, a dataset is organized in an $n \times m$ matrix, where n is the number of instances (rows) and m is the number of features (columns).

ep Learning

ep learning is the subfield of machine learning that designs and evaluates training algorithms and architectures for modern neural network models. A deep neural network is a network that has multiple (e.g., >2) layers of hidden units (or neurons).

edforward Network

feedforward network is a neural network where all the connections in the network point forward to the neurons in subsequent layer. In other words, there are no links backward from the output of a neuron to the input of a neuron in an earlier layer.

ınction

function is a deterministic mapping from a set of input values to one or more output values. In the context of machine learning, the term function is often used interchangeably with the term model.

adient Descent

adient descent is an optimization algorithm for finding a function with the minimum error with respect to modeling the patterns in a dataset. In the context of training a neural network, gradient descent is used to find the set of weights for a neuron that minimizes the error of the output of the neuron. The gradient the algorithm descends is the error gradient of the neuron as its weights are updated. The algorithm is frequently used in conjunction with backpropagation

to train neural networks with hidden layers of neurons.

PU (Graphical Processing Unit)

ecialized hardware that is optimized for fast matrix multiplication. Originally designed to increase the speed in graphics rendering but also found to speed up the training of neural networks.

STM (Long Short-Term Memory)

network designed to address the problem of vanishing gradients in recurrent neural networks. The network is composed of a cell block where activations flow through from one time-step to the next and a set of gates on the cell block that control the flow of these activations. The gates are implemented using layers of sigmoid and tanh activation functions. The standard LSTM architecture has three such gates: the forget gate, the update gate, and the output gate.

achine Learning (ML)

e field of computer science research that focuses on developing and evaluating algorithms that enable computers to learn from experience. Generally the concept of experience is represented as a dataset of historic events, and learning involves identifying and extracting useful patterns from a dataset. A machine learning algorithm takes a dataset as input and returns a model that encodes the patterns the algorithm extracted (or learned) from the data.

achine Learning Algorithm

process that analyzes a dataset and returns as model (i.e., an instantiation of a function as a computer program) that matches the patterns in the data.

odel

machine learning, a model is a computer program that encodes the patterns the machine learning algorithm has extracted from a dataset. There are many different types of machine learning models; however, deep learning is focused on creating neural network models with multiple layers of hidden neurons. A model is created (or trained) by running a machine learning algorithm on a dataset. Once the model has been trained, it can then be used to analyze new instances; the term inference is sometimes used to describe the process of analyzing a new instance using a trained model. In the context of machine

learning, the terms model and function are often used interchangeably: a model is an instantiation of a function as a computer program.

euromorphic Computing

euromorphic chips are composed of very large sets of spiking neurons architecture that are connected in a massively parallel manner.

eural Network

machine learning model that is implemented as a network of simple information processing units called neurons. It is possible to create a variety of different types of neural networks by modifying the connections between the neurons in the network. Examples of popular types of neural networks include: feedforward, convolutional, and recurrent networks.

euron

the context of deep learning (as opposed to brain science), a neuron is a simple information processing algorithm that takes a number of numeric values as input and maps these values to a high- or low-output activation. This mapping is typically implemented by first multiplying each input value by a weight, then summing the results of these multiplications, and finally passing the results of the weighted summation through an activation function.

verfitting

verfitting a dataset occurs if the model returned by a machine learning algorithm is so complex that it is able to model small variations in the data caused by the noise in the data sample.

ecurrent Neural Network

recurrent neural network has a single layer of hidden neurons, the output of which is fed back into this layer with the next input. This feedback (or recurrence) within the network gives the network a memory that enables it to process each input within the context of what it has previously processed. Recurrent neural networks are ideally suited to processing sequential or time-series data.

einforcement Learning

ıe goal of reinforcement learning is to enable an agent to learn a policy on how it should act in a given environment. A policy is a function that maps from an agent's current observations of its environment and its own internal state to an action. Typically used for online control tasks such as robot control and game playing.

ɘLU Unit

ReLU unit is a neuron that uses a rectified linear function as its activation function.

ıpervised Learning

form of machine learning where the goal is to learn a function that maps from a set of input attributes for an instance to an accurate estimate of the missing value for the target attribute of the same instance.

ırget Attribute

supervised machine learning, a target attribute is the attribute that the model is trained to estimate the value of.

ıderfitting

ıderfitting a dataset occurs if the model returned by a machine learning algorithm is too simplistic to capture the real complexity of the relationship between the inputs and outputs in a domain.

ısupervised Learning

form of machine learning where the goal is to identify regularities, such as clusters of similar instances, in the data. Unlike supervised learning, there is no target attribute in an unsupervised learning task.

ınishing Gradient

ıe vanishing gradient problem describes the fact that as more layers are added to a network it takes longer to train the network. This problem is caused by the fact that when a neural network is trained using backpropagation and gradient descent, the updating of the weights on links coming into a neuron in the network is dependent on the gradient (or sensitivity) of the network error with respect to the output of the neuron. Using backpropagation, the process of

sharing back the error gradients through a neuron involves a sequence of multiplications, often by values less than one. As a result, as the error gradient is passed back through the network, the error gradient tends to get smaller and smaller (i.e., vanish). As a direct consequence of this, the updates to weights in the early layers of the network are very small and the neurons in these layers take a long time to train.

Notes

Chapter 1

https://deepmind.com/research/alphago/.

The Elo rating system is a method for calculating the skill level of players in zero-sum games, such as Chess. It is named after its inventor, Arpad Elo.

Noise in data refers to corrupt or incorrect data. Noise in data can been caused by broken sensors, or mistakes in data entry, and so on.

By domain we mean the problem or task that we are trying to solve using machine learning. For example, it could be spam filtering, house prices prediction, or automatically classifying X-rays.

There are some scenarios where more complex dataset representations are required. For example, for time-series data, a dataset may require a three-dimensional representation, composed of a series of two-dimensional matrices, each describing the state of the system at a point in time, linked together through time. The term *tensor* generalizes the concept of a *matrix* to higher dimensions.

Chapter 2

It turns out that the relationship between annual income and happiness is linear up to a point, but that once your annual income goes beyond this point more money won’t make you happier. A study by Kahneman and Deaton (2010) found that in the US the general cutoff, after which increases in income no longer increase emotional well-being, was around $75,000.

This is the same dataset that appears in table 1.1 in chapter 1; it is repeated here for convenience.

Chapter 3

The origin is the location in a coordinate system where the axes cross. In a two-dimensional coordinate system, it is where the x-axis and y-axis cross—in other words, it is the location at coordinates x=0, y=0.

In chapter 2, we used the same approach to merge the intercept parameter of the linear model into the weights of the model.

To highlight this column organization the weights have been indexed column-row, rather than row-column.

For further discussion on the size and growth of networks, see page 23 of Goodfellow et al. 2016.

Chapter 4

Figures 3.6 and 3.7 show the linear (straight line) decision boundary of neuron that uses a threshold activation function.

This illustration of the use of associative memory for pattern completion and error correction is inspired from an example in chapter 42 of MacKay 2003.

For example, Paul Werbos's 1974 PhD thesis is credited with being the first publication to describe the use of backpropagation of errors in the training of artificial neural networks (Werbos 1974).

The Hopfield network architecture, introduced at the start of this section, also included recurrent connections (feedback loops between neurons). However, the design of the Hopfield architecture is such that a Hopfield network cannot process sequences. Consequently, it is not considered a full RNN architecture.

I originally came across this Churchland quote in Marcus 2003 (p. 25).

Critique of paper "Deep Learning Conspiracy" (*Nature* 521, p. 436), critique posted by Jürgen Schmidhuber, June 2015, available at: http://people.idsia.ch/~juergen/deep-learning-conspiracy.html.

There are a number of other ways that autoencoders can be constrained to preclude the possibility that the network will learn an uninformative identity mapping from inputs to outputs; for example, noise can be injected into the input patterns and the network can be trained to reconstruct the un-noisy data. Alternatively, the units in the hidden (or encoding) layer can be restricted to have binary values. Indeed, Hinton and his colleagues originally used networks called Restricted Boltzman Machines (RBMs) in their initial pretraining work, which used binary units in the encoding layer.

The number of layers trained during pretraining is a hyperparameter that is set based on the intuition of the data scientist and trial-and-error experimentation.

As early as 1971, Alexey Ivakhnenko's GMDH method had been shown to be able to train a deep network (up to eight layers), but this method had been largely overlooked by the research community.

. Glorot initialization is also known as Xavier initialization. Both of these names are references to one of the authors (Xavier Glorot) of the first paper that introduced this initialization procedure: Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (AISTATS), 2010, pp. 249–256.

. Glorot initialization can also be defined as sampling the weights from a Gaussian distribution with a mean of 0 and standard deviation set to the square root of 2 divided by $n_j + n_{j+1}$. However, both of these definitions of Glorot initialization have the same goal of ensuring a similar variance in activations and gradients across the layers in a network.

. https://developer.nvidia.com/cuda-zone.

Chapter 5

The explanation of LSTM units presented here is inspired by an excellent blog post by Christopher Olah, which explains LSTMs clearly and in detail; post available at: http://colah.github.io/posts/2015–08-Understanding-LSTMs/.

A sigmoid function is in fact a special case of the logistic function, and for the purposes of this discussion the distinction is not relevant.

If, for example, sigmoid units with an output range of 0 to 1 were used then activations could only be either maintained or increased at each update and eventually the cell state would become saturated with maximum values.

Chapter 6

This figure also appears in chapter 4 but it is repeated here for convenience.

Chapter 7

http://www.image-net.org.

https://en.wikipedia.org/wiki/Carver_Mead.

https://www.humanbrainproject.eu/en/.

Recitals are a non-legally binding section of a regulation that seeks to clarify the meaning of the legal text.

Laurens van der Maaten and Geoffrey Hinton, “Visualizing Data using t-SNE,” *Journal of Machine Learning Research* 9 (2008): 2579–2605.

References

zenberg, I. N., N. N. Aizenberg, and J. Vandewalles. 2000. *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications*. Springer.

ıellapilla, K., S. Puri, and Patrice Simard. 2006. “High Performance Convolutional Neural Networks for Document Processing.” In *Tenth International Workshop on Frontiers in Handwriting Recognition.*

ıurchland, P. M. 1996. *The Engine of Reason, the Seat of the Soul: A Philosophical Journey into the Brain.* MIT Press.

ɛchter, R. 1986. “Learning While Searching in Constraint-Satisfaction-Problems.” In *Proceedings of the Fifth National Conference on Artificial Intelligence* (AAAI-86), pp. 178–183.

ɛvlin, J., M. W. Chang, K. Lee, and K. Toutanova. 2018. “Bert: Pre-training of deep bidirectional transformers for language understanding.” arXiv preprint arXiv:1810.04805.

gammal, A., B. Liu, M. Elhoseiny, and M. Mazzone. 2017. “CAN: Creative Adversarial Networks, Generating ‘Art’ by Learning about Styles and Deviating from Style Norms.” arXiv:1706.07068.

ıman, J. L. 1990. “Finding Structure in Time.” *Cogn. Sci.* 14: 179–211.

id-Adar, M., I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan. 2018. “GAN-based Synthetic Medical Image Augmentation for Increased CNN Performance in Liver Lesion Classification.” arXiv:1803.01229.

kushima, K. 1980. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.” *Biol. Cybern.* 36: 193–202.

orot, X., and Y. Bengio. 2010. “Understanding the Difficulty of Training Deep Feedforward Neural Networks.” In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (AISTATS), pp. 249–256.

orot, X., A. Bordes, and Y. Bengio. 2011. “Deep Sparse Rectifier Neural

Networks.” In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (AISTATS), pp. 315–323.

ɔodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. MIT Press.

ɔodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and J. Bengio. 2014. “Generative Adversarial Nets.” In *Advances in Neural Information Processing Systems* 27: 2672–2680.

ɛ, K., X. Zhang, S. Ren, and J. Sun. 2016. “Deep Residual Learning for Image Recognition.” In *IEEE Conference on Computer Vision and Pattern Recognition* (CVPR). IEEE, pp. 770–778. https://doi.org/10.1109/CVPR.2016.90.

ɛbb, D. O. 1949. *The Organization of Behavior: A Neuropsychological Theory*. John Wiley & Sons.

ɛrculano-Houzel, S. 2009. “The Human Brain in Numbers: A Linearly Scaled-up Primate Brain.” *Front. Hum. Neurosci.* 3. https://doi.org/10.3389/neuro.09.031.2009.

nton, G. E., S. Sabour, and N. Frosst. 2018. “Matrix Capsules with EM Routing.” In *Proceedings of the 7th International Conference on Learning Representations* (ICLR).

ɔchreiter, S. 1991. Untersuchungen zu dynamischen neuronalen Netzen (Diploma). Technische Universität München.

ɔchreiter, S., Schmidhuber, J. 1997. “Long Short-Term Memory.” *Neural Comput.* 9: 1735–1780.

ɔpfield, J. J. 1982. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities.” *Proc. Natl. Acad. Sci.* 79: 2554–2558. https://doi.org/10.1073/pnas.79.8.2554.

ıbel, D. H., and T. N. Wiesel. 1962. “Receptive Fields, Binocular Interaction and Functional Architecture in the Cat’s Visual Cortex.” *J. Physiol. Lond*. 160: 106–154.

ıbel, D. H., and T. N. Wiesel. 1965. “Receptive Fields and Function Architecture in Two Nonstriate Visual Areas (18 and 19) of the Cat.” *J. Neurophysiol.* 28: 229–289.

ıkhnenko, A. G. 1971. “Polynomial Theory of Complex Systems.” *IEEE Trans.*

Syst. Man Cybern. 4: 364–378.

elleher, J. D., and B. Tierney. 2018. *Data Science.* MIT Press.

izhevsky, A., I. Sutskever, and G. E. Hinton. 2012. “Imagenet Classification with Deep Convolutional Neural Networks.” In *Advances in Neural Information Processing Systems*, pp. 1097–1105.

Cun, Y. 1989. Generalization and Network Design Strategies (Technical Report No. CRG-TR-89-4). University of Toronto Connectionist Research Group.

aas, A. L., A. Y. Hannun, and A. Y. Ng. 2013. “Rectifier Nonlinearities Improve Neural Network Acoustic Models.” In *Proceedings of the Thirteenth International Conference on Machine Learning* (ICML) Workshop on Deep Learning for Audio, Speech and Language Processing, p. 3.

acKay, D. J. C. 2003. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

arcus, G.F. 2003. *The Algebraic Mind: Integrating Connectionism and Cognitive Science*. MIT Press.

cCulloch, W. S., and W. Pitts. 1943. “A Logical Calculus of the Ideas Immanent in Nervous Activity.” *Bull. Math. Biophys.* 5: 115–133.

ikolov, T., K. Chen, G. Corrado, and J. Dean. 2013. “Efficient Estimation of Word Representations in Vector Space.” arXiv:1301.3781.

insky, M., and S. Papert. 1969. *Perceptrons*. MIT Press.

nih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. 2013. “Playing Atari with Deep Reinforcement Learning.” ArXiv13125602 Cs.

lsson, N. J. 1965. *Learning Machines: Foundations of Trainable Pattern-Classifying Systems, Series in Systems Science*. McGraw-Hill.

ı, K.-S., and K. Jung. 2004. “GPU Implementation of Neural Networks.” *Pattern Recognit.* 36: 1311–1314.

ah, C., A. Satyanarayan, I. Johnson, S. Carter, S. Ludwig, K. Ye, and A. Mordvintsev. 2018. “The Building Blocks of Interpretability.” Distill. https://doi.org/10.23915/distill.00010.

ɛagen, B., R. Adolf, P. Whatmough, G.-Y. Wei, and D. Brooks. 2017. “Deep Learning for Computer Architects.” *Synth. Lect. Comput. Archit.* 12: 1–123. https://doi.org/10.2200/S00783ED1V01Y201706CAC041.

ɛed, R. D., and R. J. Marks II. 1999. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press.

ɔsenblatt, F. 1960. On the Convergence of Reinforcement Procedures in Simple Perceptrons (Project PARA). (Report No. VG-1196-G-4). Cornell Aeronautical Laboratory, Inc., Buffalo, NY.

ɔsenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books.

ɔsenblatt, Frank, 1958. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.” *Psychol. Rev.* 65: 386–408. https://doi.org/10.1037/h0042519.

ɯmelhart, D. E., G. E. Hinton, and R. J. Williams. 1986a. “Learning Internal Representations by Error Propagation.” In D. E. Rumelhart, J. L. McClelland, and PDP Research Group, eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1. MIT Press, pp. 318–362.

ɯmelhart, D.E., J. L. McClelland, PDP Research Group, eds. 1986b. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: *Foundations*. MIT Press.

ɯmelhart, D.E., J. L. McClelland, PDP Research Group, eds. 1986c. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 2: *Psychological and Biological Models*. MIT Press.

bour, S., N. Frosst, and G. E. Hinton. 2017. “Dynamic Routing Between Capsules.” In *Proceedings of the 31st Conference on Neural Information Processing* (NIPS). pp. 3856–3866.

hmidhuber, J. 2015. “Deep Learning in Neural Networks: An Overview.” *Neural Netw.* 61: 85–117.

ɛinkraus, D., Patrice Simard, and I. Buck. 2005. “Using GPUs for Machine Learning Algorithms.” In *Eighth International Conference on Document Analysis and Recognition* (ICDAR’05). IEEE. https://doi.org/10.1109/ICDAR.2005.251.

tskever, I., O. Vinyals, and Q. V. Le. 2014. “Sequence to Sequence Learning with Neural Networks.” In *Advances in Neural Information Processing Systems* (NIPS), pp. 3104–3112.

igman, Y., M. Yang, M. Ranzato, and L. Wolf. 2014. “DeepFace: Closing the Gap to Human-Level Performance in Face Verification.” Presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1701–1708.

n der Maaten, L., and G. E. Hinton. 2008. “Visualizing Data Using t-SNE.” *J. Mach. Learn. Res.* 9, 2579–2605.

swani, A., N. Shazer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin. 2017. “Attention Is All You Need.” In *Proceedings of the 31st Conference on Neural Information Processing* (NIPS), pp. 5998–6008.

erbos, P. 1974. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.” PhD diss., Harvard University.

idrow, B., and M.E. Hoff. 1960. Adaptive Switching Circuits (Technical Report No. 1553-1). Stanford Electronics Laboratories, Stanford University, Stanford, California.

ı, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., Bengio, Y. 2015. “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention.” In *Proceedings of the 32nd International Conference on Machine Learning*, *Proceedings of Machine Learning Research*. PMLR, pp. 2048–2057.

Further Readings

Books on Deep Learning and Neural Networks

ıarniak, Eugene. 2018. *Introduction to Deep Learning*. MIT Press.

ɔodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

ıgan, Martin T., Howard B. Demuth, Mark Hudson Beale, and Orlando De Jesús. 2014. *Neural Network Design*. 2nd ed.

ɐagen, Brandon, Robert Adolf, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. “Deep Learning for Computer Architects.” *Synthesis Lectures on Computer Architecture* 12 (4): 1–123.

jnowski, Terrence J. 2018. *The Deep Learning Revolution*. MIT Press.

Online Resources

elsen, Michael A. 2015. *Neural Networks and Deep Learning*. Determination Press. Available at: http://neuralnetworksanddeeplearning.com.

still (an open access journal with many articles on deep learning and machine learning). Available at: https://distill.pub.

Overview Journal Articles

Cun, Yann, Yoshua Bengio, and Geoffrey E. Hintron. 2015. “Deep Learning.” *Nature* 521: 436–444.

hmidhuber, Jürgen. 2015. “Deep Learning in Neural Networks: An Overview.” *Neural Networks* 61: 85–117.

Index

The MIT Press Essential Knowledge Series

Auctions, Timothy P. Hubbard and Harry J. Paarsch
The Book, Amaranth Borsuk
Carbon Capture, Howard J. Herzog
Cloud Computing, Nayan B. Ruparelia
Computational Thinking, Peter J. Denning and Matti Tedre
Computing: A Concise History, Paul E. Ceruzzi
The Conscious Mind, Zoltan E. Torey
Crowdsourcing, Daren C. Brabham
Data Science, John D. Kelleher and Brendan Tierney
Deep Learning, John D. Kelleher
Extremism, J. M. Berger
Food, Fabio Parasecoli
Free Will, Mark Balaguer
The Future, Nick Montfort
GPS, Paul E. Ceruzzi
Haptics, Lynette A. Jones
Information and Society, Michael Buckland
Information and the Modern Corporation, James W. Cortada
Intellectual Property Strategy, John Palfrey
The Internet of Things, Samuel Greengard
Machine Learning: The New AI, Ethem Alpaydin
Machine Translation, Thierry Poibeau
Memes in Digital Culture, Limor Shifman
Metadata, Jeffrey Pomerantz
The Mind–Body Problem, Jonathan Westphal
MOOCs, Jonathan Haber
Neuroplasticity, Moheb Costandi
Nihilism, Nolen Gertz
Open Access, Peter Suber

Paradox, Margaret Cuonzo
Photo Authentication, Hany Farid
Post-Truth, Lee McIntyre
Robots, John Jordan
School Choice, David R. Garcia
Self-Tracking, Gina Neff and Dawn Nafus
Sexual Consent, Milena Popova
Spaceflight, Michael J. Neufeld
Sustainability, Kent E. Portney
Synesthesia, Richard E. Cytowic
The Technological Singularity, Murray Shanahan
3D Printing, John Jordan
Understanding Beliefs, Nils J. Nilsson
Waves, Frederic Raichlen

John D. Kelleher is a Professor of Computer Science and the Academic Leader of the Information, Communication and Entertainment (ICE) research institute at the Technological University Dublin (TU Dublin). He has over twenty years' experience in research and teaching in the fields of artificial intelligence, natural language processing, and machine learning. He has published more than a hundred academic articles in these fields, and two MIT Press books: *Data Science* (2018) and *Fundamentals of Machine Learning for Predictive Data Analytics* (2015). His research is supported by the ADAPT Research Centre (https://www.adaptcentre.ie), which is funded by Science Foundation Ireland (Grant 13/RC/2106) and is co-funded by the European Regional Development fund, and by PRECISE4Q project (https://precise4q.eu), which is funded through the European Union's Horizon 2020 research and innovation program under grant agreement No. 777107.