

Notation

This section provides a concise reference describing the notation used throughout this book. If you are unfamiliar with any of the corresponding mathematical concepts, we describe most of these ideas in chapters 2–4.

Numbers and Arrays

a	A scalar (integer or real)
\mathbf{a}	A vector
\mathbf{A}	A matrix
\mathbf{A}	A tensor
\mathbf{I}_n	Identity matrix with n rows and n columns
\mathbf{I}	Identity matrix with dimensionality implied by context
$\mathbf{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\mathbf{a})$	A square, diagonal matrix with diagonal entries given by \mathbf{a}
\mathbf{a}	A scalar random variable
\mathbf{a}	A vector-valued random variable
\mathbf{A}	A matrix-valued random variable

Sets and Graphs

\mathbb{A}	A set
\mathbb{R}	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \setminus \mathbb{B}$	Set subtraction, i.e., the set containing the elements of \mathbb{A} that are not in \mathbb{B}
\mathcal{G}	A graph
$Pa_{\mathcal{G}}(x_i)$	The parents of x_i in \mathcal{G}

Indexing

a_i	Element i of vector \mathbf{a} , with indexing starting at 1
a_{-i}	All elements of vector \mathbf{a} except for element i
$A_{i,j}$	Element i, j of matrix \mathbf{A}
$\mathbf{A}_{i,:}$	Row i of matrix \mathbf{A}
$\mathbf{A}_{:,i}$	Column i of matrix \mathbf{A}
$A_{i,j,k}$	Element (i, j, k) of a 3-D tensor \mathbf{A}
$\mathbf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector \mathbf{a}

Linear Algebra Operations

\mathbf{A}^T	Transpose of matrix \mathbf{A}
\mathbf{A}^+	Moore-Penrose pseudoinverse of \mathbf{A}
$\mathbf{A} \odot \mathbf{B}$	Element-wise (Hadamard) product of \mathbf{A} and \mathbf{B}
$\det(\mathbf{A})$	Determinant of \mathbf{A}

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\mathbf{x}} y$	Gradient of y with respect to \mathbf{x}
$\nabla_{\mathbf{X}} y$	Matrix derivatives of y with respect to \mathbf{X}
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to \mathbf{X}
$\frac{\partial f}{\partial \mathbf{x}}$	Jacobian matrix $\mathbf{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
$\nabla_{\mathbf{x}}^2 f(\mathbf{x})$ or $\mathbf{H}(f)(\mathbf{x})$	The Hessian matrix of f at input point \mathbf{x}
$\int f(\mathbf{x}) d\mathbf{x}$	Definite integral over the entire domain of \mathbf{x}
$\int_{\mathbb{S}} f(\mathbf{x}) d\mathbf{x}$	Definite integral with respect to \mathbf{x} over the set \mathbb{S}

Probability and Information Theory

$a \perp b$	The random variables a and b are independent
$a \perp b \mid c$	They are conditionally independent given c
$P(a)$	A probability distribution over a discrete variable
$p(a)$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$a \sim P$	Random variable a has distribution P
$\mathbb{E}_{x \sim P}[f(x)]$ or $\mathbb{E}f(x)$	Expectation of $f(x)$ with respect to $P(x)$
$\text{Var}(f(x))$	Variance of $f(x)$ under $P(x)$
$\text{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(x)$
$H(x)$	Shannon entropy of the random variable x
$D_{\text{KL}}(P \ Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over \mathbf{x} with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \rightarrow \mathbb{B}$	The function f with domain \mathbb{A} and range \mathbb{B}
$f \circ g$	Composition of the functions f and g
$f(\mathbf{x}; \boldsymbol{\theta})$	A function of \mathbf{x} parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\mathbf{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\frac{1}{1 + \exp(-x)}$
$\zeta(x)$	Softplus, $\log(1 + \exp(x))$
$\ \mathbf{x}\ _p$	L^p norm of \mathbf{x}
$\ \mathbf{x}\ $	L^2 norm of \mathbf{x}
x^+	Positive part of x , i.e., $\max(0, x)$
$\mathbf{1}_{\text{condition}}$	is 1 if the condition is true, 0 otherwise

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\mathbf{x})$, $f(\mathbf{X})$, or $f(\mathbf{X})$. This denotes the application of f to the array element-wise. For example, if $\mathbf{C} = \sigma(\mathbf{X})$, then $C_{i,j,k} = \sigma(X_{i,j,k})$ for all valid values of i , j and k .

Datasets and Distributions

p_{data}	The data generating distribution	<i>underlying "true" distribution</i>
\hat{p}_{data}	The empirical distribution defined by the training set	<i>observed distribution, empirical data</i>
\mathbb{X}	A set of training examples	<i>can be seen as an "estimate" of the true distribution, i.e. the hat</i>
$\mathbf{x}^{(i)}$	The i -th example (input) from a dataset	
$y^{(i)}$ or $\mathbf{y}^{(i)}$	The target associated with $\mathbf{x}^{(i)}$ for supervised learning	
\mathbf{X}	The $m \times n$ matrix with input example $\mathbf{x}^{(i)}$ in row $\mathbf{X}_{i,:}$	

Chapter 1

Introduction

Inventors have long dreamed of creating machines that think. This desire dates back to at least the time of ancient Greece. The mythical figures Pygmalion, Daedalus, and Hephaestus may all be interpreted as legendary inventors, and Galatea, Talos, and Pandora may all be regarded as artificial life (Ovid and Martin, 2004; Sparkes, 1996; Tandy, 1997).

When programmable computers were first conceived, people wondered whether such machines might become intelligent, over a hundred years before one was built (Lovelace, 1842). Today, **artificial intelligence** (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research.

In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers—problems that can be described by a list of formal, mathematical rules. The true challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images.

This book is about a solution to these more intuitive problems. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all the knowledge that the computer needs. The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts

are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI **deep learning**.

Many of the early successes of AI took place in relatively sterile and formal environments and did not require computers to have much knowledge about the world. For example, IBM’s Deep Blue chess-playing system defeated world champion Garry Kasparov in 1997 ([Hsu, 2002](#)). Chess is of course a very simple world, containing only sixty-four locations and thirty-two pieces that can move in only rigidly circumscribed ways. Devising a successful chess strategy is a tremendous accomplishment, but the challenge is not due to the difficulty of describing the set of chess pieces and allowable moves to the computer. Chess can be completely described by a very brief list of completely formal rules, easily provided ahead of time by the programmer.

Ironically, abstract and formal tasks that are among the most difficult mental undertakings for a human being are among the easiest for a computer. Computers have long been able to defeat even the best human chess player but only recently have begun matching some of the abilities of average human beings to recognize objects or speech. A person’s everyday life requires an immense amount of knowledge about the world. Much of this knowledge is subjective and intuitive, and therefore difficult to articulate in a formal way. Computers need to capture this same knowledge in order to behave in an intelligent way. One of the key challenges in artificial intelligence is how to get this informal knowledge into a computer.

Several artificial intelligence projects have sought to hard-code knowledge about the world in formal languages. A computer can reason automatically about statements in these formal languages using logical inference rules. This is known as the **knowledge base** approach to artificial intelligence. None of these projects has led to a major success. One of the most famous such projects is Cyc ([Lenat and Guha, 1989](#)). Cyc is an inference engine and a database of statements in a language called CycL. These statements are entered by a staff of human supervisors. It is an unwieldy process. People struggle to devise formal rules with enough complexity to accurately describe the world. For example, Cyc failed to understand a story about a person named Fred shaving in the morning ([Linde, 1992](#)). Its inference engine detected an inconsistency in the story: it knew that people do not have electrical parts, but because Fred was holding an electric razor, it believed the entity “FredWhileShaving” contained electrical parts. It therefore asked whether Fred was still a person while he was shaving.

The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting

patterns from raw data. This capability is known as **machine learning**. The introduction of machine learning enabled computers to tackle problems involving knowledge of the real world and make decisions that appear subjective. A simple machine learning algorithm called **logistic regression** can determine whether to recommend cesarean delivery (Mor-Yosef *et al.*, 1990). A simple machine learning algorithm called **naive Bayes** can separate legitimate e-mail from spam e-mail.

The performance of these simple machine learning algorithms depends heavily on the **representation** of the data they are given. For example, when logistic regression is used to recommend cesarean delivery, the AI system does not examine the patient directly. Instead, the doctor tells the system several pieces of relevant information, such as the presence or absence of a uterine scar. Each piece of information included in the representation of the patient is known as a **feature**. Logistic regression learns how each of these features of the patient correlates with various outcomes. However, it cannot influence how features are defined in any way. If logistic regression were given an MRI scan of the patient, rather than the doctor's formalized report, it would not be able to make useful predictions. Individual pixels in an MRI scan have negligible correlation with any complications that might occur during delivery.

This dependence on representations is a general phenomenon that appears throughout computer science and even daily life. In computer science, operations such as searching a collection of data can proceed exponentially faster if the collection is structured and indexed intelligently. People can easily perform arithmetic on Arabic numerals but find arithmetic on Roman numerals much more time consuming. It is not surprising that the choice of representation has an enormous effect on the performance of machine learning algorithms. For a simple visual example, see figure 1.1.

Many artificial intelligence tasks can be solved by designing the right set of features to extract for that task, then providing these features to a simple machine learning algorithm. For example, a useful feature for speaker identification from sound is an estimate of the size of the speaker's vocal tract. This feature gives a strong clue as to whether the speaker is a man, woman, or child.

For many tasks, however, it is difficult to know what features should be extracted. For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape, but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the

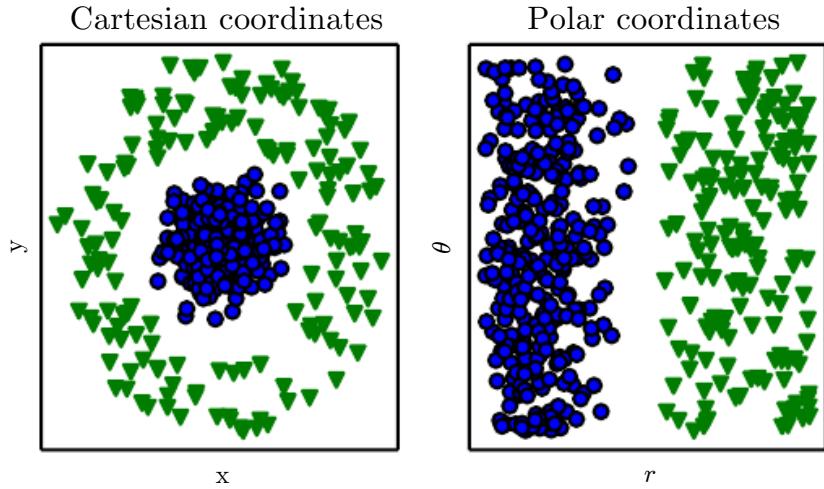


Figure 1.1: Example of different representations: suppose we want to separate two categories of data by drawing a line between them in a scatterplot. In the plot on the left, we represent some data using Cartesian coordinates, and the task is impossible. In the plot on the right, we represent the data with polar coordinates and the task becomes simple to solve with a vertical line. (Figure produced in collaboration with David Warde-Farley.)

foreground obscuring part of the wheel, and so on.

One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as **representation learning**. Learned representations often result in much better performance than can be obtained with hand-designed representations. They also enable AI systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a simple task in minutes, or for a complex task in hours to months. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.

The quintessential example of a representation learning algorithm is the **autoencoder**. An autoencoder is the combination of an **encoder** function, which converts the input data into a different representation, and a **decoder** function, which converts the new representation back into the original format. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but they are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties.

When designing features or algorithms for learning features, our goal is usually to separate the **factors of variation** that explain the observed data. In this

context, we use the word “factors” simply to refer to separate sources of influence; the factors are usually not combined by multiplication. Such factors are often not quantities that are directly observed. Instead, they may exist as either unobserved objects or unobserved forces in the physical world that affect observable quantities. They may also exist as constructs in the human mind that provide useful simplifying explanations or inferred causes of the observed data. They can be thought of as concepts or abstractions that help us make sense of the rich variability in the data. When analyzing a speech recording, the factors of variation include the speaker’s age, their sex, their accent and the words they are speaking. When analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun.

A major source of difficulty in many real-world artificial intelligence applications is that many of the factors of variation influence every single piece of data we are able to observe. The individual pixels in an image of a red car might be very close to black at night. The shape of the car’s silhouette depends on the viewing angle. Most applications require us to *disentangle* the factors of variation and discard the ones that we do not care about.

Of course, it can be very difficult to extract such high-level, abstract features from raw data. Many of these factors of variation, such as a speaker’s accent, can be identified only using sophisticated, nearly human-level understanding of the data. When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us.

Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning enables the computer to build complex concepts out of simpler concepts. Figure 1.2 shows how a deep learning system can represent the concept of an image of a person by combining simpler concepts, such as corners and contours, which are in turn defined in terms of edges.

The quintessential example of a deep learning model is the feedforward deep network, or **multilayer perceptron** (MLP). A multilayer perceptron is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions. We can think of each application of a different mathematical function as providing a new representation of the input.

The idea of learning the right representation for the data provides one perspective on deep learning. Another perspective on deep learning is that depth enables the computer to learn a multistep computer program. Each layer of the representation can be thought of as the state of the computer’s memory after

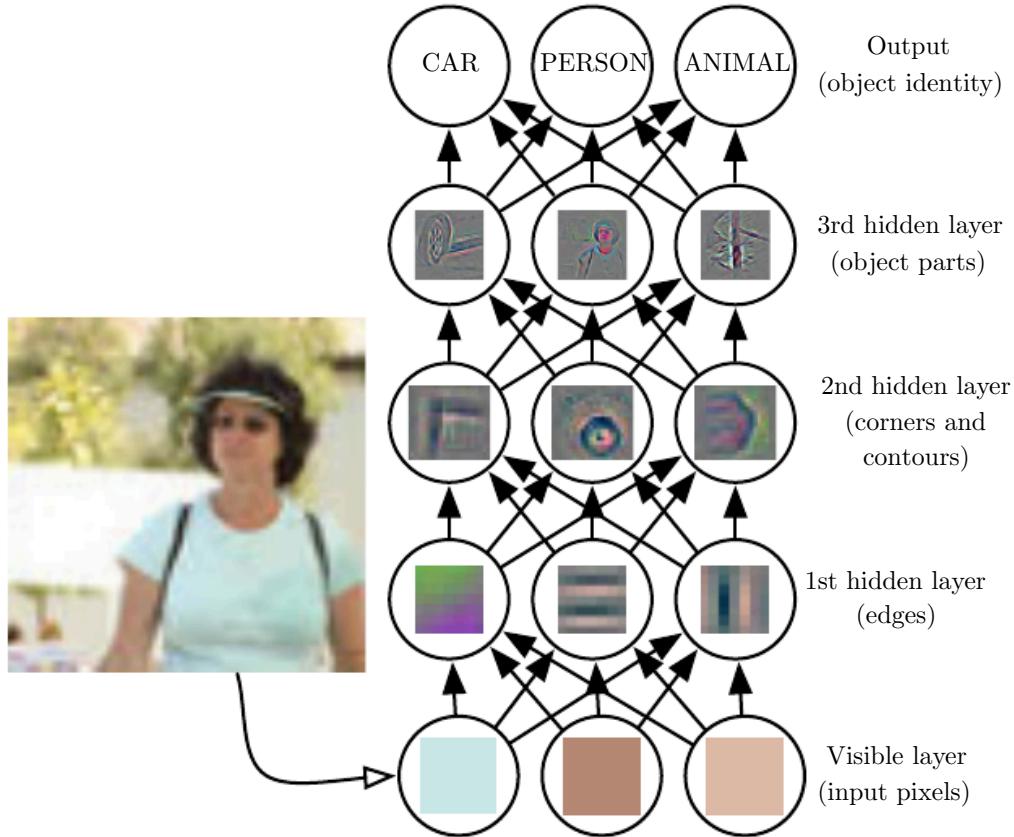


Figure 1.2: Illustration of a deep learning model. It is difficult for a computer to understand the meaning of raw sensory input data, such as this image represented as a collection of pixel values. The function mapping from a set of pixels to an object identity is very complicated. Learning or evaluating this mapping seems insurmountable if tackled directly. Deep learning resolves this difficulty by breaking the desired complicated mapping into a series of nested simple mappings, each described by a different layer of the model. The input is presented at the **visible layer**, so named because it contains the variables that we are able to observe. Then a series of **hidden layers** extracts increasingly abstract features from the image. These layers are called “hidden” because their values are not given in the data; instead the model must determine which concepts are useful for explaining the relationships in the observed data. The images here are visualizations of the kind of feature represented by each hidden unit. Given the pixels, the first layer can easily identify edges, by comparing the brightness of neighboring pixels. Given the first hidden layer’s description of the edges, the second hidden layer can easily search for corners and extended contours, which are recognizable as collections of edges. Given the second hidden layer’s description of the image in terms of corners and contours, the third hidden layer can detect entire parts of specific objects, by finding specific collections of contours and corners. Finally, this description of the image in terms of the object parts it contains can be used to recognize the objects present in the image. Images reproduced with permission from [Zeiler and Fergus \(2014\)](#).

executing another set of instructions in parallel. Networks with greater depth can execute more instructions in sequence. Sequential instructions offer great power because later instructions can refer back to the results of earlier instructions. According to this view of deep learning, not all the information in a layer’s activations necessarily encodes factors of variation that explain the input. The representation also stores state information that helps to execute a program that can make sense of the input. This state information could be analogous to a counter or pointer in a traditional computer program. It has nothing to do with the content of the input specifically, but it helps the model to organize its processing.

There are two main ways of measuring the depth of a model. The first view is based on the number of sequential instructions that must be executed to evaluate the architecture. We can think of this as the length of the longest path through a flow chart that describes how to compute each of the model’s outputs given its inputs. Just as two equivalent computer programs will have different lengths depending on which language the program is written in, the same function may be drawn as a flowchart with different depths depending on which functions we allow to be used as individual steps in the flowchart. Figure 1.3 illustrates how this choice of language can give two different measurements for the same architecture.

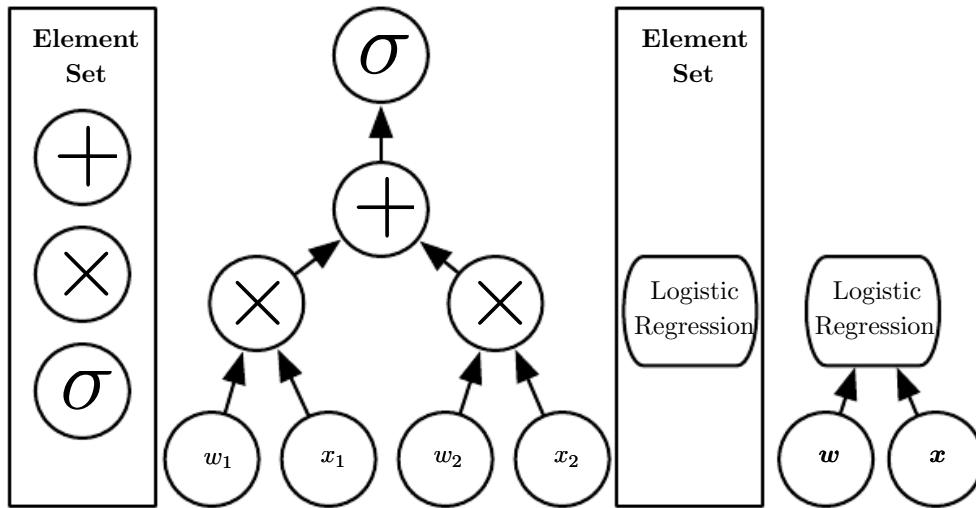


Figure 1.3: Illustration of computational graphs mapping an input to an output where each node performs an operation. Depth is the length of the longest path from input to output but depends on the definition of what constitutes a possible computational step. The computation depicted in these graphs is the output of a logistic regression model, $\sigma(\mathbf{w}^T \mathbf{x})$, where σ is the logistic sigmoid function. If we use addition, multiplication and logistic sigmoid as the elements of our computer language, then this model has depth three. If we view logistic regression as an element itself, then this model has depth one.

Another approach, used by deep probabilistic models, regards the depth of a model as being not the depth of the computational graph but the depth of the graph describing how concepts are related to each other. In this case, the depth of the flowchart of the computations needed to compute the representation of each concept may be much deeper than the graph of the concepts themselves. This is because the system’s understanding of the simpler concepts can be refined given information about the more complex concepts. For example, an AI system observing an image of a face with one eye in shadow may initially see only one eye. After detecting that a face is present, the system can then infer that a second eye is probably present as well. In this case, the graph of concepts includes only two layers—a layer for eyes and a layer for faces—but the graph of computations includes $2n$ layers if we refine our estimate of each concept given the other n times.

Because it is not always clear which of these two views—the depth of the computational graph, or the depth of the probabilistic modeling graph—is most relevant, and because different people choose different sets of smallest elements from which to construct their graphs, there is no single correct value for the depth of an architecture, just as there is no single correct value for the length of a computer program. Nor is there a consensus about how much depth a model requires to qualify as “deep.” However, deep learning can be safely regarded as the study of models that involve a greater amount of composition of either learned functions or learned concepts than traditional machine learning does.

To summarize, deep learning, the subject of this book, is an approach to AI. Specifically, it is a type of machine learning, a technique that enables computer systems to improve with experience and data. We contend that machine learning is the only viable approach to building AI systems that can operate in complicated real-world environments. Deep learning is a particular kind of machine learning that achieves great power and flexibility by representing the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones. Figure 1.4 illustrates the relationship between these different AI disciplines. Figure 1.5 gives a high-level schematic of how each works.

1.1 Who Should Read This Book?

This book can be useful for a variety of readers, but we wrote it with two target audiences in mind. One of these target audiences is university students (undergraduate or graduate) learning about machine learning, including those who are beginning a career in deep learning and artificial intelligence research. The other

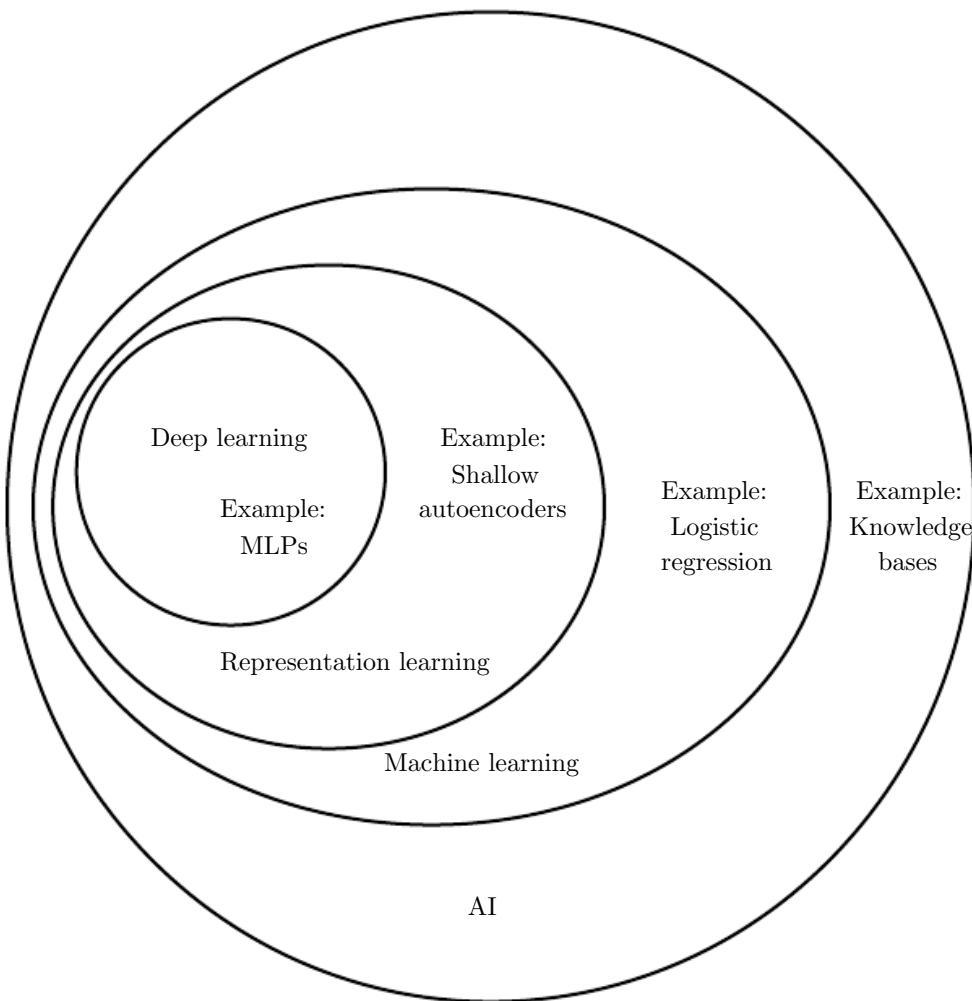


Figure 1.4: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

target audience is software engineers who do not have a machine learning or statistics background but want to rapidly acquire one and begin using deep learning in their product or platform. Deep learning has already proved useful in many software disciplines, including computer vision, speech and audio processing, natural language processing, robotics, bioinformatics and chemistry, video games, search engines, online advertising and finance.

This book has been organized into three parts to best accommodate a variety of readers. Part I introduces basic mathematical tools and machine learning concepts. Part II describes the most established deep learning algorithms, which are essentially solved technologies. Part III describes more speculative ideas that are widely believed to be important for future research in deep learning.

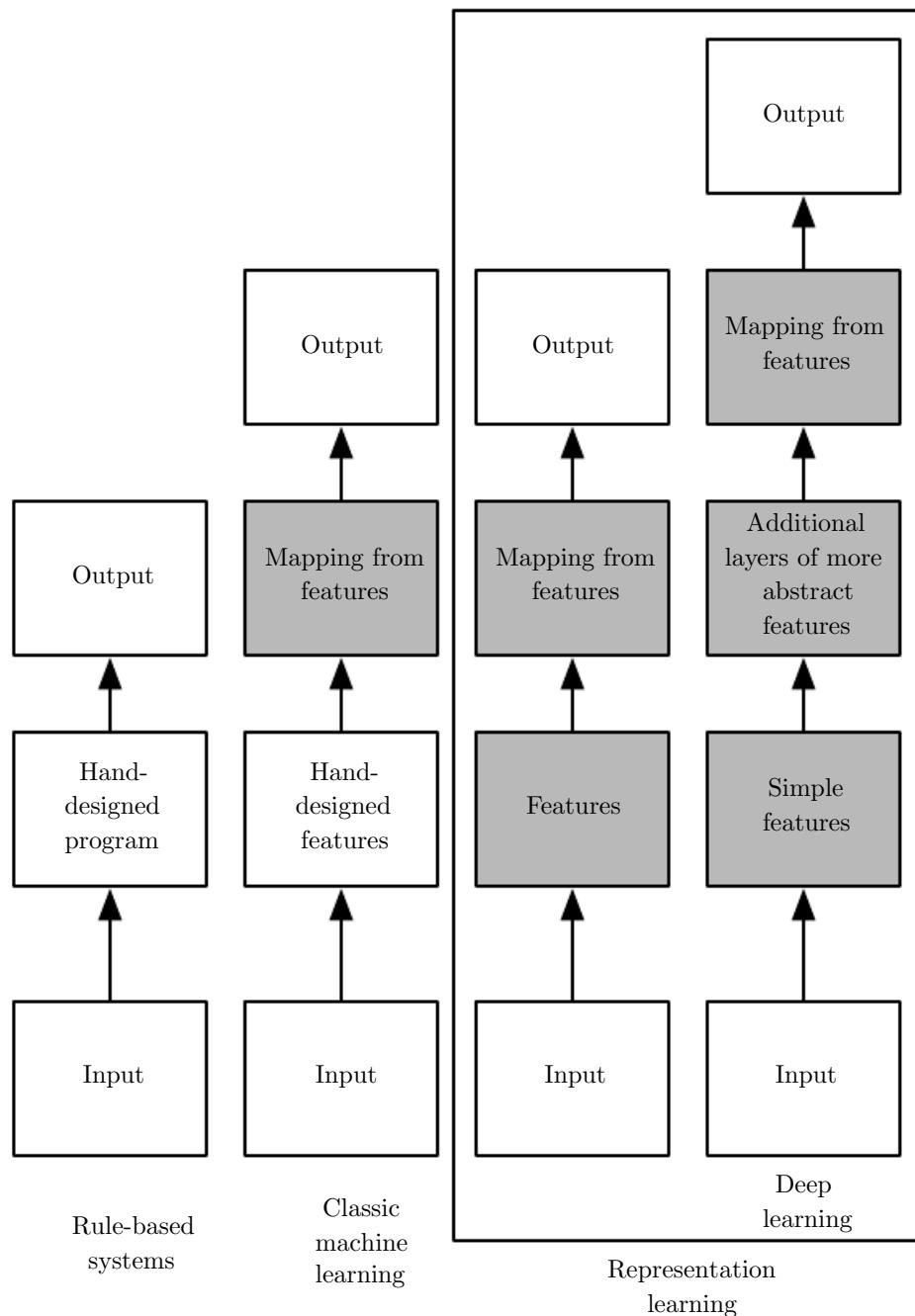


Figure 1.5: Flowcharts showing the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data.

Readers should feel free to skip parts that are not relevant given their interests or background. Readers familiar with linear algebra, probability, and fundamental machine learning concepts can skip part I, for example, while those who just want to implement a working system need not read beyond part II. To help choose which

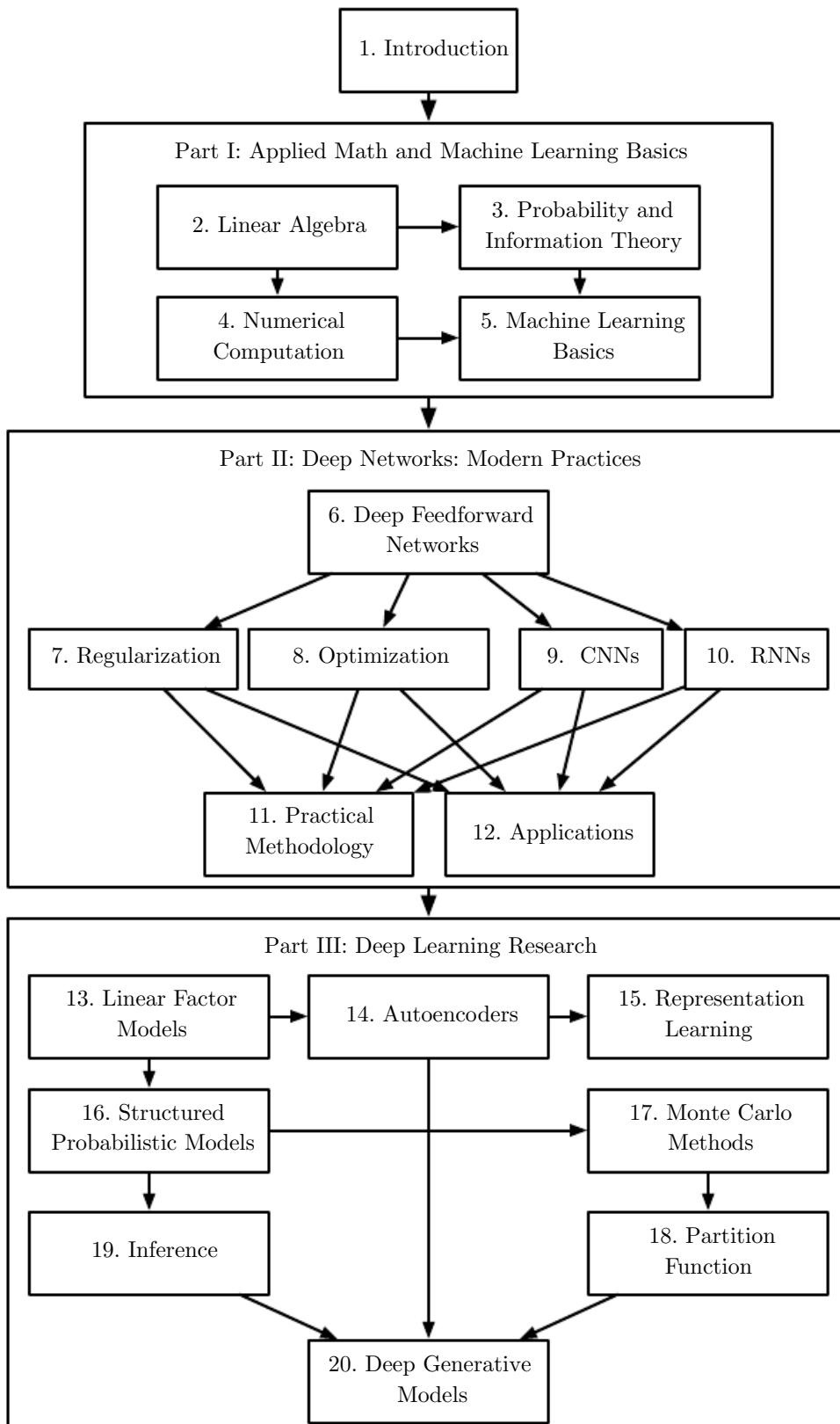


Figure 1.6: The high-level organization of the book. An arrow from one chapter to another indicates that the former chapter is prerequisite material for understanding the latter.

chapters to read, figure 1.6 provides a flowchart showing the high-level organization of the book.

We do assume that all readers come from a computer science background. We assume familiarity with programming, a basic understanding of computational performance issues, complexity theory, introductory level calculus and some of the terminology of graph theory.

1.2 Historical Trends in Deep Learning

It is easiest to understand deep learning with some historical context. Rather than providing a detailed history of deep learning, we identify a few key trends:

- Deep learning has had a long and rich history, but has gone by many names, reflecting different philosophical viewpoints, and has waxed and waned in popularity.
- Deep learning has become more useful as the amount of available training data has increased.
- Deep learning models have grown in size over time as computer infrastructure (both hardware and software) for deep learning has improved.
- Deep learning has solved increasingly complicated applications with increasing accuracy over time.

1.2.1 The Many Names and Changing Fortunes of Neural Networks

We expect that many readers of this book have heard of deep learning as an exciting new technology, and are surprised to see a mention of “history” in a book about an emerging field. In fact, deep learning dates back to the 1940s. Deep learning only *appears* to be new, because it was relatively unpopular for several years preceding its current popularity, and because it has gone through many different names, only recently being called “deep learning.” The field has been rebranded many times, reflecting the influence of different researchers and different perspectives.

A comprehensive history of deep learning is beyond the scope of this textbook. Some basic context, however, is useful for understanding deep learning. Broadly speaking, there have been three waves of development: deep learning known as **cybernetics** in the 1940s–1960s, deep learning known as **connectionism** in the

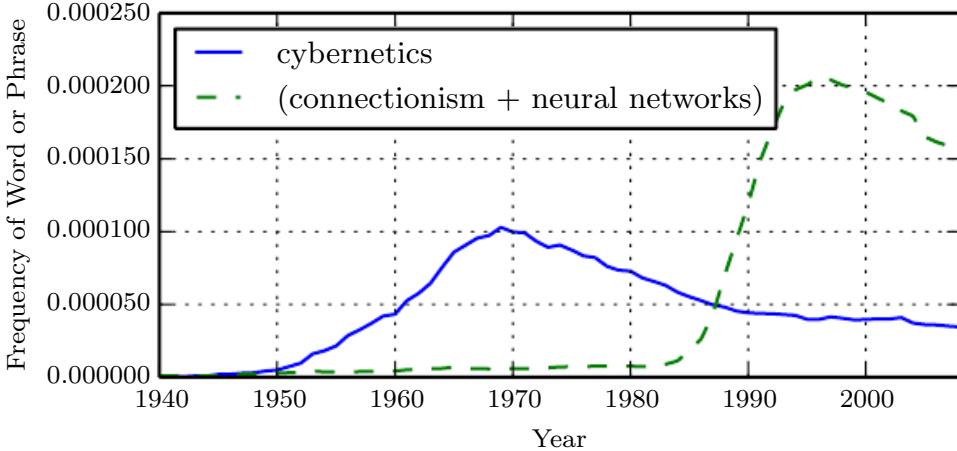


Figure 1.7: Two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases “cybernetics” and “connectionism” or “neural networks,” according to Google Books (the third wave is too recent to appear). The first wave started with cybernetics in the 1940s–1960s, with the development of theories of biological learning (McCulloch and Pitts, 1943; Hebb, 1949) and implementations of the first models, such as the perceptron (Rosenblatt, 1958), enabling the training of a single neuron. The second wave started with the connectionist approach of the 1980–1995 period, with back-propagation (Rumelhart *et al.*, 1986a) to train a neural network with one or two hidden layers. The current and third wave, deep learning, started around 2006 (Hinton *et al.*, 2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007a) and is just now appearing in book form as of 2016. The other two waves similarly appeared in book form much later than the corresponding scientific activity occurred.

1980s–1990s, and the current resurgence under the name deep learning beginning in 2006. This is quantitatively illustrated in figure 1.7.

Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, that is, models of how learning happens or could happen in the brain. As a result, one of the names that deep learning has gone by is **artificial neural networks** (ANNs). The corresponding perspective on deep learning models is that they are engineered systems inspired by the biological brain (whether the human brain or the brain of another animal). While the kinds of neural networks used for machine learning have sometimes been used to understand brain function (Hinton and Shallice, 1991), they are generally not designed to be realistic models of biological function. The neural perspective on deep learning is motivated by two main ideas. One idea is that the brain provides a proof by example that intelligent behavior is possible, and a conceptually straightforward path to building intelligence is to reverse engineer the computational principles behind the brain and duplicate its functionality. Another

perspective is that it would be deeply interesting to understand the brain and the principles that underlie human intelligence, so machine learning models that shed light on these basic scientific questions are useful apart from their ability to solve engineering applications.

The modern term “deep learning” goes beyond the neuroscientific perspective on the current breed of machine learning models. It appeals to a more general principle of learning *multiple levels of composition*, which can be applied in machine learning frameworks that are not necessarily neurally inspired.

The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective. These models were designed to take a set of n input values x_1, \dots, x_n and associate them with an output y . These models would learn a set of weights w_1, \dots, w_n and compute their output $f(\mathbf{x}, \mathbf{w}) = x_1 w_1 + \dots + x_n w_n$. This first wave of neural networks research was known as cybernetics, as illustrated in figure 1.7.

The McCulloch-Pitts neuron (McCulloch and Pitts, 1943) was an early model of brain function. This linear model could recognize two different categories of inputs by testing whether $f(\mathbf{x}, \mathbf{w})$ is positive or negative. Of course, for the model to correspond to the desired definition of the categories, the weights needed to be set correctly. These weights could be set by the human operator. In the 1950s, the perceptron (Rosenblatt, 1958, 1962) became the first model that could learn the weights that defined the categories given examples of inputs from each category. The **adaptive linear element** (ADALINE), which dates from about the same time, simply returned the value of $f(\mathbf{x})$ itself to predict a real number (Widrow and Hoff, 1960) and could also learn to predict these numbers from data.

These simple learning algorithms greatly affected the modern landscape of machine learning. The training algorithm used to adapt the weights of the ADALINE was a special case of an algorithm called **stochastic gradient descent**. Slightly modified versions of the stochastic gradient descent algorithm remain the dominant training algorithms for deep learning models today.

Models based on the $f(\mathbf{x}, \mathbf{w})$ used by the perceptron and ADALINE are called **linear models**. These models remain some of the most widely used machine learning models, though in many cases they are *trained* in different ways than the original models were trained.

Linear models have many limitations. Most famously, they cannot learn the XOR function, where $f([0, 1], \mathbf{w}) = 1$ and $f([1, 0], \mathbf{w}) = 1$ but $f([1, 1], \mathbf{w}) = 0$ and $f([0, 0], \mathbf{w}) = 0$. Critics who observed these flaws in linear models caused a backlash against biologically inspired learning in general (Minsky and Papert, 1969). This was the first major dip in the popularity of neural networks.

Today, neuroscience is regarded as an important source of inspiration for deep learning researchers, but it is no longer the predominant guide for the field.

The main reason for the diminished role of neuroscience in deep learning research today is that we simply do not have enough information about the brain to use it as a guide. To obtain a deep understanding of the actual algorithms used by the brain, we would need to be able to monitor the activity of (at the very least) thousands of interconnected neurons simultaneously. Because we are not able to do this, we are far from understanding even some of the most simple and well-studied parts of the brain (Olshausen and Field, 2005).

Neuroscience has given us a reason to hope that a single deep learning algorithm can solve many different tasks. Neuroscientists have found that ferrets can learn to “see” with the auditory processing region of their brain if their brains are rewired to send visual signals to that area (Von Melchner *et al.*, 2000). This suggests that much of the mammalian brain might use a single algorithm to solve most of the different tasks that the brain solves. Before this hypothesis, machine learning research was more fragmented, with different communities of researchers studying natural language processing, vision, motion planning and speech recognition. Today, these application communities are still separate, but it is common for deep learning research groups to study many or even all these application areas simultaneously.

We are able to draw some rough guidelines from neuroscience. The basic idea of having many computational units that become intelligent only via their interactions with each other is inspired by the brain. The neocognitron (Fukushima, 1980) introduced a powerful model architecture for processing images that was inspired by the structure of the mammalian visual system and later became the basis for the modern convolutional network (LeCun *et al.*, 1998b), as we will see in section 9.10. Most neural networks today are based on a model neuron called the **rectified linear unit**. The original cognitron (Fukushima, 1975) introduced a more complicated version that was highly inspired by our knowledge of brain function. The simplified modern version was developed incorporating ideas from many viewpoints, with Nair and Hinton (2010) and Glorot *et al.* (2011a) citing neuroscience as an influence, and Jarrett *et al.* (2009) citing more engineering-oriented influences. While neuroscience is an important source of inspiration, it need not be taken as a rigid guide. We know that actual neurons compute very different functions than modern rectified linear units, but greater neural realism has not yet led to an improvement in machine learning performance. Also, while neuroscience has successfully inspired several neural network *architectures*, we do not yet know enough about biological learning for neuroscience to offer much guidance for the *learning algorithms* we use to train these architectures.

Media accounts often emphasize the similarity of deep learning to the brain. While it is true that deep learning researchers are more likely to cite the brain as an influence than researchers working in other machine learning fields, such as kernel machines or Bayesian statistics, one should not view deep learning as an attempt to simulate the brain. Modern deep learning draws inspiration from many fields, especially applied math fundamentals like linear algebra, probability, information theory, and numerical optimization. While some deep learning researchers cite neuroscience as an important source of inspiration, others are not concerned with neuroscience at all.

It is worth noting that the effort to understand how the brain works on an algorithmic level is alive and well. This endeavor is primarily known as “computational neuroscience” and is a separate field of study from deep learning. It is common for researchers to move back and forth between both fields. The field of deep learning is primarily concerned with how to build computer systems that are able to successfully solve tasks requiring intelligence, while the field of computational neuroscience is primarily concerned with building more accurate models of how the brain actually works.

In the 1980s, the second wave of neural network research emerged in great part via a movement called **connectionism**, or **parallel distributed processing** (Rumelhart *et al.*, 1986c; McClelland *et al.*, 1995). Connectionism arose in the context of cognitive science. Cognitive science is an interdisciplinary approach to understanding the mind, combining multiple different levels of analysis. During the early 1980s, most cognitive scientists studied models of symbolic reasoning. Despite their popularity, symbolic models were difficult to explain in terms of how the brain could actually implement them using neurons. The connectionists began to study models of cognition that could actually be grounded in neural implementations (Touretzky and Minton, 1985), reviving many ideas dating back to the work of psychologist Donald Hebb in the 1940s (Hebb, 1949).

The central idea in connectionism is that a large number of simple computational units can achieve intelligent behavior when networked together. This insight applies equally to neurons in biological nervous systems as it does to hidden units in computational models.

Several key concepts arose during the connectionism movement of the 1980s that remain central to today’s deep learning.

One of these concepts is that of **distributed representation** (Hinton *et al.*, 1986). This is the idea that each input to a system should be represented by many features, and each feature should be involved in the representation of many possible inputs. For example, suppose we have a vision system that can recognize

cars, trucks, and birds, and these objects can each be red, green, or blue. One way of representing these inputs would be to have a separate neuron or hidden unit that activates for each of the nine possible combinations: red truck, red car, red bird, green truck, and so on. This requires nine different neurons, and each neuron must independently learn the concept of color and object identity. One way to improve on this situation is to use a distributed representation, with three neurons describing the color and three neurons describing the object identity. This requires only six neurons total instead of nine, and the neuron describing redness is able to learn about redness from images of cars, trucks and birds, not just from images of one specific category of objects. The concept of distributed representation is central to this book and is described in greater detail in chapter 15.

Another major accomplishment of the connectionist movement was the successful use of back-propagation to train deep neural networks with internal representations and the popularization of the back-propagation algorithm (Rumelhart *et al.*, 1986a; LeCun, 1987). This algorithm has waxed and waned in popularity but, as of this writing, is the dominant approach to training deep models.

During the 1990s, researchers made important advances in modeling sequences with neural networks. Hochreiter (1991) and Bengio *et al.* (1994) identified some of the fundamental mathematical difficulties in modeling long sequences, described in section 10.7. Hochreiter and Schmidhuber (1997) introduced the long short-term memory (LSTM) network to resolve some of these difficulties. Today, the LSTM is widely used for many sequence modeling tasks, including many natural language processing tasks at Google.

The second wave of neural networks research lasted until the mid-1990s. Ventures based on neural networks and other AI technologies began to make unrealistically ambitious claims while seeking investments. When AI research did not fulfill these unreasonable expectations, investors were disappointed. Simultaneously, other fields of machine learning made advances. Kernel machines (Boser *et al.*, 1992; Cortes and Vapnik, 1995; Schölkopf *et al.*, 1999) and graphical models (Jordan, 1998) both achieved good results on many important tasks. These two factors led to a decline in the popularity of neural networks that lasted until 2007.

During this time, neural networks continued to obtain impressive performance on some tasks (LeCun *et al.*, 1998b; Bengio *et al.*, 2001). The Canadian Institute for Advanced Research (CIFAR) helped to keep neural networks research alive via its Neural Computation and Adaptive Perception (NCAP) research initiative. This program united machine learning research groups led by Geoffrey Hinton at University of Toronto, Yoshua Bengio at University of Montreal, and Yann LeCun at New York University. The multidisciplinary CIFAR NCAP research initiative

also included neuroscientists and experts in human and computer vision.

At this point, deep networks were generally believed to be very difficult to train. We now know that algorithms that have existed since the 1980s work quite well, but this was not apparent circa 2006. The issue is perhaps simply that these algorithms were too computationally costly to allow much experimentation with the hardware available at the time.

The third wave of neural networks research began with a breakthrough in 2006. Geoffrey Hinton showed that a kind of neural network called a deep belief network could be efficiently trained using a strategy called greedy layer-wise pretraining (Hinton *et al.*, 2006), which we describe in more detail in section 15.1. The other CIFAR-affiliated research groups quickly showed that the same strategy could be used to train many other kinds of deep networks (Bengio *et al.*, 2007; Ranzato *et al.*, 2007a) and systematically helped to improve generalization on test examples. This wave of neural networks research popularized the use of the term “deep learning” to emphasize that researchers were now able to train deeper neural networks than had been possible before, and to focus attention on the theoretical importance of depth (Bengio and LeCun, 2007; Delalleau and Bengio, 2011; Pascanu *et al.*, 2014a; Montufar *et al.*, 2014). At this time, deep neural networks outperformed competing AI systems based on other machine learning technologies as well as hand-designed functionality. This third wave of popularity of neural networks continues to the time of this writing, though the focus of deep learning research has changed dramatically within the time of this wave. The third wave began with a focus on new unsupervised learning techniques and the ability of deep models to generalize well from small datasets, but today there is more interest in much older supervised learning algorithms and the ability of deep models to leverage large labeled datasets.

1.2.2 Increasing Dataset Sizes

One may wonder why deep learning has only recently become recognized as a crucial technology even though the first experiments with artificial neural networks were conducted in the 1950s. Deep learning has been successfully used in commercial applications since the 1990s but was often regarded as being more of an art than a technology and something that only an expert could use, until recently. It is true that some skill is required to get good performance from a deep learning algorithm. Fortunately, the amount of skill required reduces as the amount of training data increases. The learning algorithms reaching human performance on complex tasks today are nearly identical to the learning algorithms that struggled to solve toy problems in the 1980s, though the models we train with these algorithms have

undergone changes that simplify the training of very deep architectures. The most important new development is that today we can provide these algorithms with the resources they need to succeed. Figure 1.8 shows how the size of benchmark datasets has expanded remarkably over time. This trend is driven by the increasing digitization of society. As more and more of our activities take place on computers, more and more of what we do is recorded. As our computers are increasingly networked together, it becomes easier to centralize these records and curate them into a dataset appropriate for machine learning applications. The age of “Big Data”

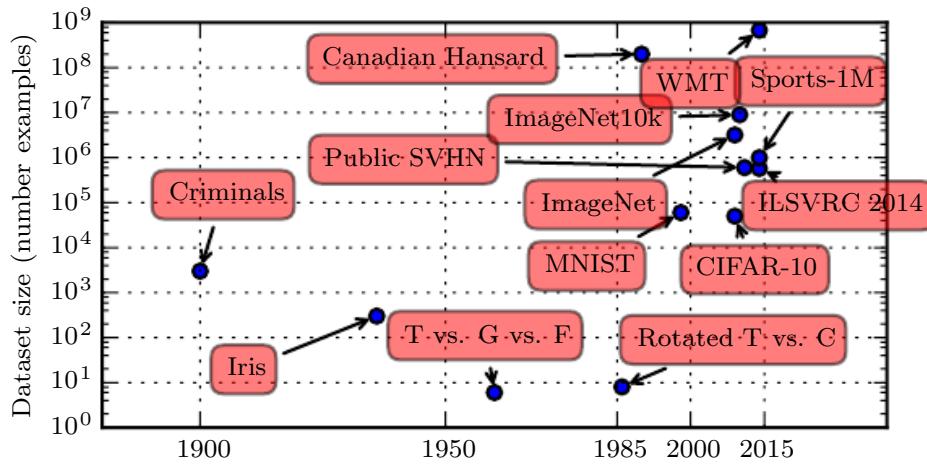


Figure 1.8: Increasing dataset size over time. In the early 1900s, statisticians studied datasets using hundreds or thousands of manually compiled measurements (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). In the 1950s through the 1980s, the pioneers of biologically inspired machine learning often worked with small synthetic datasets, such as low-resolution bitmaps of letters, that were designed to incur low computational cost and demonstrate that neural networks were able to learn specific kinds of functions (Widrow and Hoff, 1960; Rumelhart *et al.*, 1986b). In the 1980s and 1990s, machine learning became more statistical and began to leverage larger datasets containing tens of thousands of examples, such as the MNIST dataset (shown in figure 1.9) of scans of handwritten numbers (LeCun *et al.*, 1998b). In the first decade of the 2000s, more sophisticated datasets of this same size, such as the CIFAR-10 dataset (Krizhevsky and Hinton, 2009), continued to be produced. Toward the end of that decade and throughout the first half of the 2010s, significantly larger datasets, containing hundreds of thousands to tens of millions of examples, completely changed what was possible with deep learning. These datasets included the public Street View House Numbers dataset (Netzer *et al.*, 2011), various versions of the ImageNet dataset (Deng *et al.*, 2009, 2010a; Russakovsky *et al.*, 2014a), and the Sports-1M dataset (Karpathy *et al.*, 2014). At the top of the graph, we see that datasets of translated sentences, such as IBM’s dataset constructed from the Canadian Hansard (Brown *et al.*, 1990) and the WMT 2014 English to French dataset (Schwenk, 2014), are typically far ahead of other dataset sizes.

8	9	0	1	2	3	4	7	8	9	0	1	2	3	4	5	6	7	8	6
4	2	6	4	7	5	5	4	7	8	9	2	9	3	9	3	8	2	0	5
0	1	0	4	2	6	5	3	5	3	8	0	0	3	4	1	5	3	0	8
3	0	6	2	7	1	1	8	1	7	1	3	8	9	7	6	7	4	1	6
7	5	1	7	1	9	8	0	6	9	4	9	9	3	7	1	9	2	2	5
3	7	8	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	0
1	2	3	4	5	6	7	8	9	8	1	0	5	5	1	9	0	4	1	9
3	8	4	7	7	8	5	0	6	5	5	3	3	3	9	8	1	4	0	6
1	0	0	6	2	1	1	3	2	8	8	7	8	4	6	0	2	0	3	6
8	7	1	5	9	9	3	2	4	9	4	6	5	3	2	8	5	9	4	1
6	5	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
8	9	0	1	2	3	4	5	6	7	8	9	6	4	2	6	4	7	5	5
4	7	8	9	2	9	3	9	3	8	2	0	9	8	0	5	6	0	1	0
4	2	6	5	5	5	4	3	4	1	5	3	0	8	3	0	6	2	7	1
1	8	1	7	1	3	8	5	4	2	0	9	7	6	7	4	1	6	8	4
7	5	1	2	6	7	1	9	8	0	6	9	4	9	9	6	2	3	7	1
9	2	2	5	3	7	8	0	1	2	3	4	5	6	7	8	0	1	2	3
4	5	6	7	8	0	1	2	3	4	5	6	7	8	9	2	1	2	1	3
9	9	8	5	3	7	0	7	7	5	7	9	9	4	7	0	3	4	1	4
4	7	5	8	1	4	8	4	1	8	6	6	4	6	3	5	7	2	5	9

Figure 1.9: Example inputs from the MNIST dataset. The “NIST” stands for National Institute of Standards and Technology, the agency that originally collected this data. The “M” stands for “modified,” since the data has been preprocessed for easier use with machine learning algorithms. The MNIST dataset consists of scans of handwritten digits and associated labels describing which digit 0–9 is contained in each image. This simple classification problem is one of the simplest and most widely used tests in deep learning research. It remains popular despite being quite easy for modern techniques to solve. Geoffrey Hinton has described it as “the *drosophila* of machine learning,” meaning that it enables machine learning researchers to study their algorithms in controlled laboratory conditions, much as biologists often study fruit flies.

has made machine learning much easier because the key burden of statistical estimation—generalizing well to new data after observing only a small amount of data—has been considerably lightened. As of 2016, a rough rule of thumb is that a supervised deep learning algorithm will generally achieve acceptable performance with around 5,000 labeled examples per category and will match or

exceed human performance when trained with a dataset containing at least 10 million labeled examples. Working successfully with datasets smaller than this is an important research area, focusing in particular on how we can take advantage of large quantities of unlabeled examples, with unsupervised or semi-supervised learning.

1.2.3 Increasing Model Sizes

Another key reason that neural networks are wildly successful today after enjoying comparatively little success since the 1980s is that we have the computational resources to run much larger models today. One of the main insights of connectionism is that animals become intelligent when many of their neurons work together. An individual neuron or small collection of neurons is not particularly useful.

Biological neurons are not especially densely connected. As seen in figure 1.10, our machine learning models have had a number of connections per neuron within an order of magnitude of even mammalian brains for decades.

In terms of the total number of neurons, neural networks have been astonishingly small until quite recently, as shown in figure 1.11. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. This growth is driven by faster computers with larger memory and by the availability of larger datasets. Larger networks are able to achieve higher accuracy on more complex tasks. This trend looks set to continue for decades. Unless new technologies enable faster scaling, artificial neural networks will not have the same number of neurons as the human brain until at least the 2050s. Biological neurons may represent more complicated functions than current artificial neurons, so biological neural networks may be even larger than this plot portrays.

In retrospect, it is not particularly surprising that neural networks with fewer neurons than a leech were unable to solve sophisticated artificial intelligence problems. Even today’s networks, which we consider quite large from a computational systems point of view, are smaller than the nervous system of even relatively primitive vertebrate animals like frogs.

The increase in model size over time, due to the availability of faster CPUs, the advent of general purpose GPUs (described in section 12.1.2), faster network connectivity and better software infrastructure for distributed computing, is one of the most important trends in the history of deep learning. This trend is generally expected to continue well into the future.

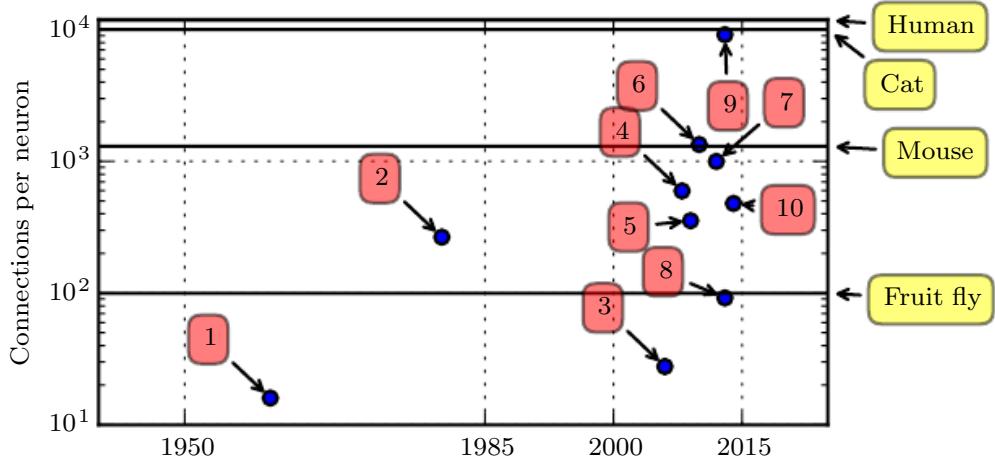


Figure 1.10: Number of connections per neuron over time. Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, and it is quite common for other neural networks to have as many connections per neuron as smaller mammals like mice. Even the human brain does not have an exorbitant amount of connections per neuron. Biological neural network sizes from [Wikipedia \(2015\)](#).

1. Adaptive linear element ([Widrow and Hoff, 1960](#))
2. Neocognitron ([Fukushima, 1980](#))
3. GPU-accelerated convolutional network ([Chellapilla et al., 2006](#))
4. Deep Boltzmann machine ([Salakhutdinov and Hinton, 2009a](#))
5. Unsupervised convolutional network ([Jarrett et al., 2009](#))
6. GPU-accelerated multilayer perceptron ([Ciresan et al., 2010](#))
7. Distributed autoencoder ([Le et al., 2012](#))
8. Multi-GPU convolutional network ([Krizhevsky et al., 2012](#))
9. COTS HPC unsupervised convolutional network ([Coates et al., 2013](#))
10. GoogLeNet ([Szegedy et al., 2014a](#))

1.2.4 Increasing Accuracy, Complexity and Real-World Impact

Since the 1980s, deep learning has consistently improved in its ability to provide accurate recognition and prediction. Moreover, deep learning has consistently been applied with success to broader and broader sets of applications.

The earliest deep models were used to recognize individual objects in tightly cropped, extremely small images ([Rumelhart et al., 1986a](#)). Since then there has been a gradual increase in the size of images neural networks could process. Modern object recognition networks process rich high-resolution photographs and do not

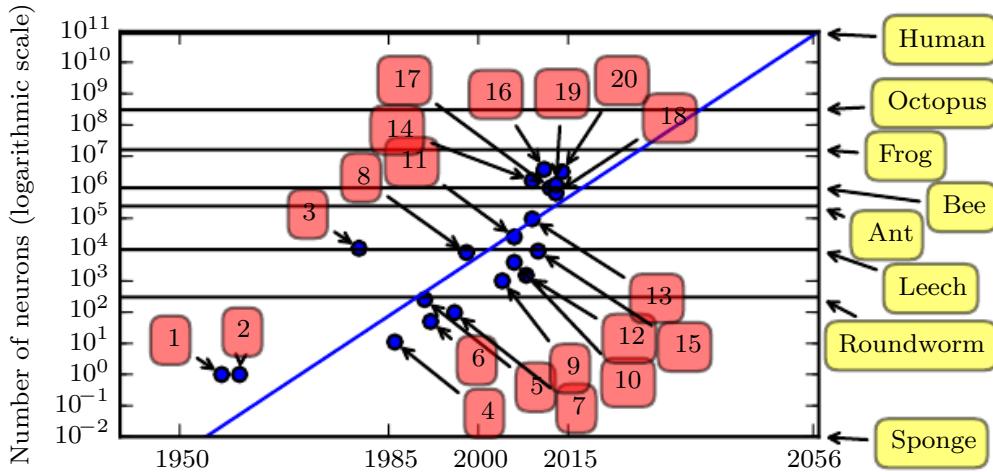


Figure 1.11: Increasing neural network size over time. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart *et al.*, 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio *et al.*, 1991)
7. Mean field sigmoid belief network (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton *et al.*, 2006)
11. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009)
15. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le *et al.*, 2012)
18. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

have a requirement that the photo be cropped near the object to be recognized (Krizhevsky *et al.*, 2012). Similarly, the earliest networks could recognize only two kinds of objects (or in some cases, the absence or presence of a single kind of object), while these modern networks typically recognize at least 1,000 different categories of objects. The largest contest in object recognition is the ImageNet

Large Scale Visual Recognition Challenge (ILSVRC) held each year. A dramatic moment in the meteoric rise of deep learning came when a convolutional network won this challenge for the first time and by a wide margin, bringing down the state-of-the-art top-5 error rate from 26.1 percent to 15.3 percent (Krizhevsky *et al.*, 2012), meaning that the convolutional network produces a ranked list of possible categories for each image, and the correct category appeared in the first five entries of this list for all but 15.3 percent of the test examples. Since then, these competitions are consistently won by deep convolutional nets, and as of this writing, advances in deep learning have brought the latest top-5 error rate in this contest down to 3.6 percent, as shown in figure 1.12.

Deep learning has also had a dramatic impact on speech recognition. After improving throughout the 1990s, the error rates for speech recognition stagnated starting in about 2000. The introduction of deep learning (Dahl *et al.*, 2010; Deng *et al.*, 2010b; Seide *et al.*, 2011; Hinton *et al.*, 2012a) to speech recognition resulted in a sudden drop in error rates, with some error rates cut in half. We explore this history in more detail in section 12.3.

Deep networks have also had spectacular successes for pedestrian detection and image segmentation (Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013) and yielded superhuman performance in traffic sign classification (Ciresan *et al.*, 2012).

At the same time that the scale and accuracy of deep networks have increased,

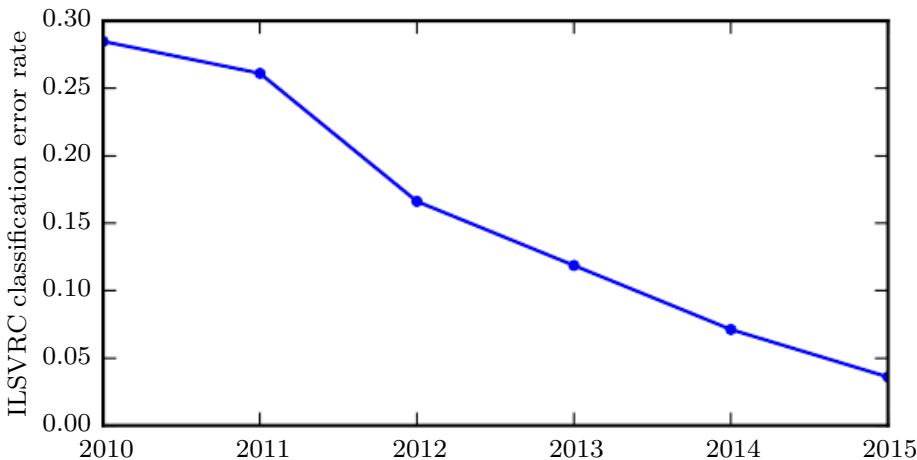


Figure 1.12: Decreasing error rate over time. Since deep networks reached the scale necessary to compete in the ImageNet Large Scale Visual Recognition Challenge, they have consistently won the competition every year, yielding lower and lower error rates each time. Data from Russakovsky *et al.* (2014b) and He *et al.* (2015).

so has the complexity of the tasks that they can solve. Goodfellow *et al.* (2014d) showed that neural networks could learn to output an entire sequence of characters transcribed from an image, rather than just identifying a single object. Previously, it was widely believed that this kind of learning required labeling of the individual elements of the sequence (Gülçehre and Bengio, 2013). Recurrent neural networks, such as the LSTM sequence model mentioned above, are now used to model relationships between *sequences* and other *sequences* rather than just fixed inputs. This sequence-to-sequence learning seems to be on the cusp of revolutionizing another application: machine translation (Sutskever *et al.*, 2014; Bahdanau *et al.*, 2015).

This trend of increasing complexity has been pushed to its logical conclusion with the introduction of neural Turing machines (Graves *et al.*, 2014) that learn to read from memory cells and write arbitrary content to memory cells. Such neural networks can learn simple programs from examples of desired behavior. For example, they can learn to sort lists of numbers given examples of scrambled and sorted sequences. This self-programming technology is in its infancy, but in the future it could in principle be applied to nearly any task.

Another crowning achievement of deep learning is its extension to the domain of **reinforcement learning**. In the context of reinforcement learning, an autonomous agent must learn to perform a task by trial and error, without any guidance from the human operator. DeepMind demonstrated that a reinforcement learning system based on deep learning is capable of learning to play Atari video games, reaching human-level performance on many tasks (Mnih *et al.*, 2015). Deep learning has also significantly improved the performance of reinforcement learning for robotics (Finn *et al.*, 2015).

Many of these applications of deep learning are highly profitable. Deep learning is now used by many top technology companies, including Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA, and NEC.

Advances in deep learning have also depended heavily on advances in software infrastructure. Software libraries such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012), PyLearn2 (Goodfellow *et al.*, 2013c), Torch (Collobert *et al.*, 2011b), DistBelief (Dean *et al.*, 2012), Caffe (Jia, 2013), MXNet (Chen *et al.*, 2015), and TensorFlow (Abadi *et al.*, 2015) have all supported important research projects or commercial products.

Deep learning has also made contributions to other sciences. Modern convolutional networks for object recognition provide a model of visual processing that neuroscientists can study (DiCarlo, 2013). Deep learning also provides useful tools for processing massive amounts of data and making useful predictions in scientific

fields. It has been successfully used to predict how molecules will interact in order to help pharmaceutical companies design new drugs (Dahl *et al.*, 2014), to search for subatomic particles (Baldi *et al.*, 2014), and to automatically parse microscope images used to construct a 3-D map of the human brain (Knowles-Barley *et al.*, 2014). We expect deep learning to appear in more and more scientific fields in the future.

In summary, deep learning is an approach to machine learning that has drawn heavily on our knowledge of the human brain, statistics and applied math as it developed over the past several decades. In recent years, deep learning has seen tremendous growth in its popularity and usefulness, largely as the result of more powerful computers, larger datasets and techniques to train deeper networks. The years ahead are full of challenges and opportunities to improve deep learning even further and to bring it to new frontiers.

Chapter 4

Numerical Computation

Machine learning algorithms usually require a high amount of numerical computation. This typically refers to algorithms that solve mathematical problems by methods that update estimates of the solution via an iterative process, rather than analytically deriving a formula to provide a symbolic expression for the correct solution. Common operations include optimization (finding the value of an argument that minimizes or maximizes a function) and solving systems of linear equations. Even just evaluating a mathematical function on a digital computer can be difficult when the function involves real numbers, which cannot be represented precisely using a finite amount of memory.

4.1 Overflow and Underflow

The fundamental difficulty in performing continuous math on a digital computer is that we need to represent infinitely many real numbers with a finite number of bit patterns. This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer. In many cases, this is just rounding error. Rounding error is problematic, especially when it compounds across many operations, and can cause algorithms that work in theory to fail in practice if they are not designed to minimize the accumulation of rounding error.

One form of rounding error that is particularly devastating is **underflow**. Underflow occurs when numbers near zero are rounded to zero. Many functions behave qualitatively differently when their argument is zero rather than a small positive number. For example, we usually want to avoid division by zero (some

software environments will raise exceptions when this occurs, others will return a result with a placeholder not-a-number value) or taking the logarithm of zero (this is usually treated as $-\infty$, which then becomes not-a-number if it is used for many further arithmetic operations).

Another highly damaging form of numerical error is **overflow**. Overflow occurs when numbers with large magnitude are approximated as ∞ or $-\infty$. Further arithmetic will usually change these infinite values into not-a-number values.

One example of a function that must be stabilized against underflow and overflow is the **softmax function**. The softmax function is often used to predict the probabilities associated with a multinoulli distribution. The softmax function is defined to be

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}. \quad (4.1)$$

Consider what happens when all the x_i are equal to some constant c . Analytically, we can see that all the outputs should be equal to $\frac{1}{n}$. Numerically, this may not occur when c has large magnitude. If c is very negative, then $\exp(c)$ will underflow. This means the denominator of the softmax will become 0, so the final result is undefined. When c is very large and positive, $\exp(c)$ will overflow, again resulting in the expression as a whole being undefined. Both of these difficulties can be resolved by instead evaluating $\text{softmax}(\mathbf{z})$ where $\mathbf{z} = \mathbf{x} - \max_i x_i$. Simple algebra shows that the value of the softmax function is not changed analytically by adding or subtracting a scalar from the input vector. Subtracting $\max_i x_i$ results in the largest argument to \exp being 0, which rules out the possibility of overflow. Likewise, at least one term in the denominator has a value of 1, which rules out the possibility of underflow in the denominator leading to a division by zero.

There is still one small problem. Underflow in the numerator can still cause the expression as a whole to evaluate to zero. This means that if we implement $\log \text{softmax}(\mathbf{x})$ by first running the softmax subroutine then passing the result to the \log function, we could erroneously obtain $-\infty$. Instead, we must implement a separate function that calculates $\log \text{softmax}$ in a numerically stable way. The $\log \text{softmax}$ function can be stabilized using the same trick as we used to stabilize the softmax function.

For the most part, we do not explicitly detail all the numerical considerations involved in implementing the various algorithms described in this book. Developers of low-level libraries should keep numerical issues in mind when implementing deep learning algorithms. Most readers of this book can simply rely on low-level libraries that provide stable implementations. In some cases, it is possible to implement a new algorithm and have the new implementation automatically

stabilized. Theano ([Bergstra et al., 2010](#); [Bastien et al., 2012](#)) is an example of a software package that automatically detects and stabilizes many common numerically unstable expressions that arise in the context of deep learning.

4.2 Poor Conditioning

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

Consider the function $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$. When $\mathbf{A} \in \mathbb{R}^{n \times n}$ has an eigenvalue decomposition, its **condition number** is

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \quad (4.2)$$

This is the ratio of the magnitude of the largest and smallest eigenvalue. When this number is large, matrix inversion is particularly sensitive to error in the input.

This sensitivity is an intrinsic property of the matrix itself, not the result of rounding error during matrix inversion. Poorly conditioned matrices amplify pre-existing errors when we multiply by the true matrix inverse. In practice, the error will be compounded further by numerical errors in the inversion process itself.

4.3 Gradient-Based Optimization

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(\mathbf{x})$ by altering \mathbf{x} . We usually phrase most optimization problems in terms of minimizing $f(\mathbf{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\mathbf{x})$.

The function we want to minimize or maximize is called the **objective function**, or **criterion**. When we are minimizing it, we may also call it the **cost function**, **loss function**, or **error function**. In this book, we use these terms interchangeably, though some machine learning publications assign special meaning to some of these terms.

We often denote the value that minimizes or maximizes a function with a superscript $*$. For example, we might say $\mathbf{x}^* = \arg \min f(\mathbf{x})$.

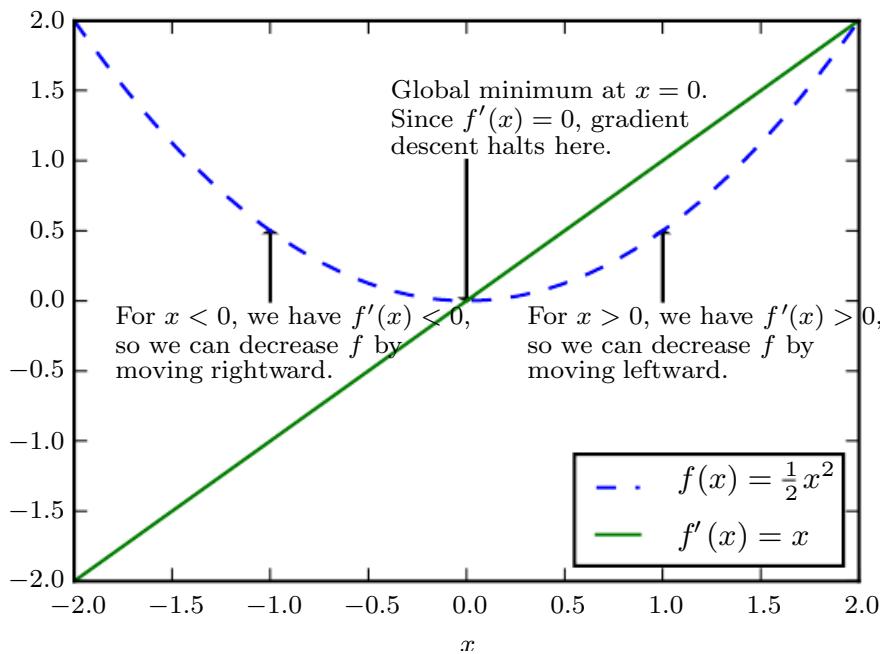


Figure 4.1: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

We assume the reader is already familiar with calculus but provide a brief review of how calculus concepts relate to optimization here.

Suppose we have a function $y = f(x)$, where both x and y are real numbers. The **derivative** of this function is denoted as $f'(x)$ or as $\frac{dy}{dx}$. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x . In other words, it specifies how to scale a small change in the input to obtain the corresponding change in the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y . For example, we know that $f(x - \epsilon \text{ sign}(f'(x)))$ is less than $f(x)$ for small enough ϵ . We can thus reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative. This technique is called **gradient descent** (Cauchy, 1847). See figure 4.1 for an example of this technique.

When $f'(x) = 0$, the derivative provides no information about which direction to move. Points where $f'(x) = 0$ are known as **critical points**, or **stationary points**. A **local minimum** is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A **local maximum** is a point where $f(x)$ is higher than at all neighboring points,

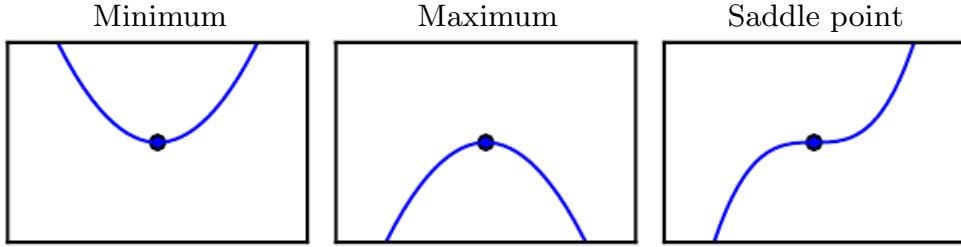


Figure 4.2: Types of critical points. Examples of the three types of critical points in one dimension. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points; a local maximum, which is higher than the neighboring points; or a saddle point, which has neighbors that are both higher and lower than the point itself.

so it is not possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither maxima nor minima. These are known as **saddle points**. See figure 4.2 for examples of each type of critical point.

A point that obtains the absolute lowest value of $f(x)$ is a **global minimum**. There can be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally optimal. In the context of deep learning, we optimize functions that may have many local minima that are not optimal and many saddle points surrounded by very flat regions. All of this makes optimization difficult, especially when the input to the function is multidimensional. We therefore usually settle for finding a value of f that is very low but not necessarily minimal in any formal sense. See figure 4.3 for an example.

We often minimize functions that have multiple inputs: $f : \mathbb{R}^n \rightarrow \mathbb{R}$. For the concept of “minimization” to make sense, there must still be only one (scalar) output.

For functions with multiple inputs, we must make use of the concept of **partial derivatives**. The partial derivative $\frac{\partial}{\partial x_i} f(\mathbf{x})$ measures how f changes as only the variable x_i increases at point \mathbf{x} . The **gradient** generalizes the notion of derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all the partial derivatives, denoted $\nabla_{\mathbf{x}} f(\mathbf{x})$. Element i of the gradient is the partial derivative of f with respect to x_i . In multiple dimensions,

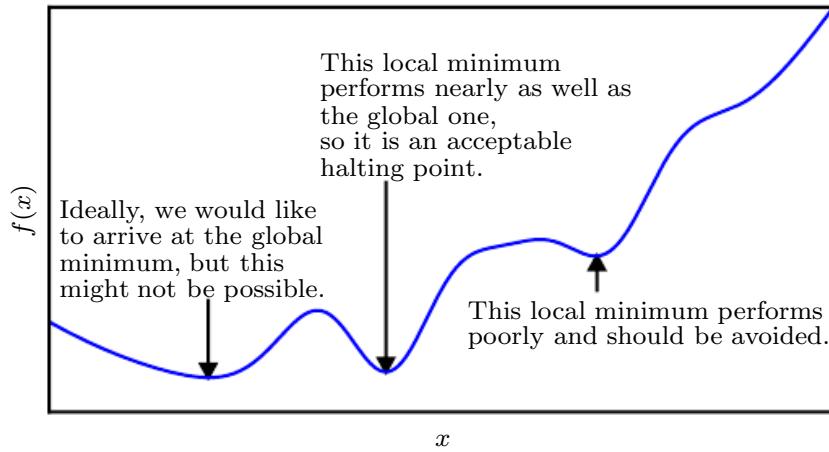


Figure 4.3: Approximate minimization. Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function.

critical points are points where every element of the gradient is equal to zero.

The **directional derivative** in direction \mathbf{u} (a unit vector) is the slope of the function f in direction \mathbf{u} . In other words, the directional derivative is the derivative of the function $f(\mathbf{x} + \alpha\mathbf{u})$ with respect to α , evaluated at $\alpha = 0$. Using the chain rule, we can see that $\frac{\partial}{\partial \alpha} f(\mathbf{x} + \alpha\mathbf{u})$ evaluates to $\mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$ when $\alpha = 0$.

To minimize f , we would like to find the direction in which f decreases the fastest. We can do this using the directional derivative:

$$\min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u}=1} \mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (4.3)$$

$$= \min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u}=1} \|\mathbf{u}\|_2 \|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 \cos \theta \quad (4.4)$$

where θ is the angle between \mathbf{u} and the gradient. Substituting in $\|\mathbf{u}\|_2 = 1$ and ignoring factors that do not depend on \mathbf{u} , this simplifies to $\min_{\mathbf{u}} \cos \theta$. This is minimized when \mathbf{u} points in the opposite direction as the gradient. In other words, the gradient points directly uphill, and the negative gradient points directly downhill. We can decrease f by moving in the direction of the negative gradient. This is known as the **method of steepest descent**, or **gradient descent**.

Steepest descent proposes a new point

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (4.5)$$

where ϵ is the **learning rate**, a positive scalar determining the size of the step. We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f(\mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}))$ for several values of ϵ and choose the one that results in the smallest objective function value. This last strategy is called a **line search**.

Steepest descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we may be able to avoid running this iterative algorithm and just jump directly to the critical point by solving the equation $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$ for \mathbf{x} .

Although gradient descent is limited to optimization in continuous spaces, the general concept of repeatedly making a small move (that is approximately the best small move) toward better configurations can be generalized to discrete spaces. Ascending an objective function of discrete parameters is called **hill climbing** (Russel and Norvig, 2003).

4.3.1 Beyond the Gradient: Jacobian and Hessian Matrices

Sometimes we need to find all the partial derivatives of a function whose input and output are both vectors. The matrix containing all such partial derivatives is known as a **Jacobian matrix**. Specifically, if we have a function $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, then the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{n \times m}$ of \mathbf{f} is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$.

We are also sometimes interested in a derivative of a derivative. This is known as a **second derivative**. For example, for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the derivative with respect to x_i of the derivative of f with respect to x_j is denoted as $\frac{\partial^2}{\partial x_i \partial x_j} f$. In a single dimension, we can denote $\frac{d^2}{dx^2} f$ by $f''(x)$. The second derivative tells us how the first derivative will change as we vary the input. This is important because it tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone. We can think of the second derivative as measuring **curvature**. Suppose we have a quadratic function (many functions that arise in practice are not quadratic but can be approximated well as quadratic, at least locally). If such a function has a second derivative of zero, then there is no curvature. It is a perfectly flat line, and its value can be predicted using only the gradient. If the gradient is 1, then we can make a step of size ϵ along the negative gradient, and the cost function will decrease by ϵ . If the second derivative is negative, the function curves downward, so the cost function will actually decrease by more than ϵ . Finally, if the second derivative is positive, the function curves upward, so the cost function can decrease by less than ϵ . See

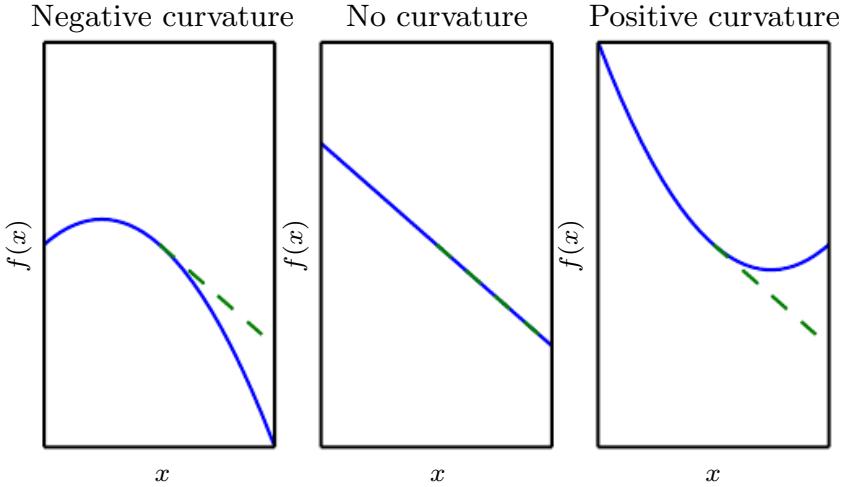


Figure 4.4: The second derivative determines the curvature of a function. Here we show quadratic functions with various curvature. The dashed line indicates the value of the cost function we would expect based on the gradient information alone as we make a gradient step downhill. With negative curvature, the cost function actually decreases faster than the gradient predicts. With no curvature, the gradient predicts the decrease correctly. With positive curvature, the function decreases more slowly than expected and eventually begins to increase, so steps that are too large can actually increase the function inadvertently.

figure 4.4 to see how different forms of curvature affect the relationship between the value of the cost function predicted by the gradient and the true value.

When our function has multiple input dimensions, there are many second derivatives. These derivatives can be collected together into a matrix called the **Hessian matrix**. The Hessian matrix $\mathbf{H}(f)(\mathbf{x})$ is defined such that

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}). \quad (4.6)$$

Equivalently, the Hessian is the Jacobian of the gradient.

Anywhere that the second partial derivatives are continuous, the differential operators are commutative; that is, their order can be swapped:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\mathbf{x}). \quad (4.7)$$

This implies that $H_{i,j} = H_{j,i}$, so the Hessian matrix is symmetric at such points. Most of the functions we encounter in the context of deep learning have a symmetric Hessian almost everywhere. Because the Hessian matrix is real and symmetric, we can decompose it into a set of real eigenvalues and an orthogonal basis of

eigenvectors. The second derivative in a specific direction represented by a unit vector \mathbf{d} is given by $\mathbf{d}^\top \mathbf{H} \mathbf{d}$. When \mathbf{d} is an eigenvector of \mathbf{H} , the second derivative in that direction is given by the corresponding eigenvalue. For other directions of \mathbf{d} , the directional second derivative is a weighted average of all the eigenvalues, with weights between 0 and 1, and eigenvectors that have a smaller angle with \mathbf{d} receiving more weight. The maximum eigenvalue determines the maximum second derivative, and the minimum eigenvalue determines the minimum second derivative.

The (directional) second derivative tells us how well we can expect a gradient descent step to perform. We can make a second-order Taylor series approximation to the function $f(\mathbf{x})$ around the current point $\mathbf{x}^{(0)}$:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{g} + \frac{1}{2} (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H} (\mathbf{x} - \mathbf{x}^{(0)}), \quad (4.8)$$

where \mathbf{g} is the gradient and \mathbf{H} is the Hessian at $\mathbf{x}^{(0)}$. If we use a learning rate of ϵ , then the new point \mathbf{x} will be given by $\mathbf{x}^{(0)} - \epsilon \mathbf{g}$. Substituting this into our approximation, we obtain

$$f(\mathbf{x}^{(0)} - \epsilon \mathbf{g}) \approx f(\mathbf{x}^{(0)}) - \epsilon \mathbf{g}^\top \mathbf{g} + \frac{1}{2} \epsilon^2 \mathbf{g}^\top \mathbf{H} \mathbf{g}. \quad (4.9)$$

There are three terms here: the original value of the function, the expected improvement due to the slope of the function, and the correction we must apply to account for the curvature of the function. When this last term is too large, the gradient descent step can actually move uphill. When $\mathbf{g}^\top \mathbf{H} \mathbf{g}$ is zero or negative, the Taylor series approximation predicts that increasing ϵ forever will decrease f forever. In practice, the Taylor series is unlikely to remain accurate for large ϵ , so one must resort to more heuristic choices of ϵ in this case. When $\mathbf{g}^\top \mathbf{H} \mathbf{g}$ is positive, solving for the optimal step size that decreases the Taylor series approximation of the function the most yields

$$\epsilon^* = \frac{\mathbf{g}^\top \mathbf{g}}{\mathbf{g}^\top \mathbf{H} \mathbf{g}}. \quad (4.10)$$

In the worst case, when \mathbf{g} aligns with the eigenvector of \mathbf{H} corresponding to the maximal eigenvalue λ_{\max} , then this optimal step size is given by $\frac{1}{\lambda_{\max}}$. To the extent that the function we minimize can be approximated well by a quadratic function, the eigenvalues of the Hessian thus determine the scale of the learning rate.

The second derivative can be used to determine whether a critical point is a local maximum, a local minimum, or a saddle point. Recall that on a critical point, $f'(x) = 0$. When the second derivative $f''(x) > 0$, the first derivative $f'(x)$

increases as we move to the right and decreases as we move to the left. This means $f'(x - \epsilon) < 0$ and $f'(x + \epsilon) > 0$ for small enough ϵ . In other words, as we move right, the slope begins to point uphill to the right, and as we move left, the slope begins to point uphill to the left. Thus, when $f'(x) = 0$ and $f''(x) > 0$, we can conclude that x is a local minimum. Similarly, when $f'(x) = 0$ and $f''(x) < 0$, we can conclude that x is a local maximum. This is known as the **second derivative test**. Unfortunately, when $f''(x) = 0$, the test is inconclusive. In this case x may be a saddle point or a part of a flat region.

In multiple dimensions, we need to examine all the second derivatives of the function. Using the eigendecomposition of the Hessian matrix, we can generalize the second derivative test to multiple dimensions. At a critical point, where $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$, we can examine the eigenvalues of the Hessian to determine whether the critical point is a local maximum, local minimum, or saddle point. When the Hessian is positive definite (all its eigenvalues are positive), the point is a local minimum. This can be seen by observing that the directional second derivative in any direction must be positive, and making reference to the univariate second derivative test. Likewise, when the Hessian is negative definite (all its eigenvalues are negative), the point is a local maximum. In multiple dimensions, it is actually possible to find positive evidence of saddle points in some cases. When at least one eigenvalue is positive and at least one eigenvalue is negative, we know that \mathbf{x} is a local maximum on one cross section of f but a local minimum on another cross section. See figure 4.5 for an example. Finally, the multidimensional second derivative test can be inconclusive, just as the univariate version can. The test is inconclusive whenever all the nonzero eigenvalues have the same sign but at least one eigenvalue is zero. This is because the univariate second derivative test is inconclusive in the cross section corresponding to the zero eigenvalue.

In multiple dimensions, there is a different second derivative for each direction at a single point. The condition number of the Hessian at this point measures how much the second derivatives differ from each other. When the Hessian has a poor condition number, gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change in the derivative, so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer. Poor condition number also makes choosing a good step size difficult. The step size must be small enough to avoid overshooting the minimum and going uphill in directions with strong positive curvature. This usually means that the step size is too small to make significant progress in other directions with less curvature. See figure 4.6 for an example.

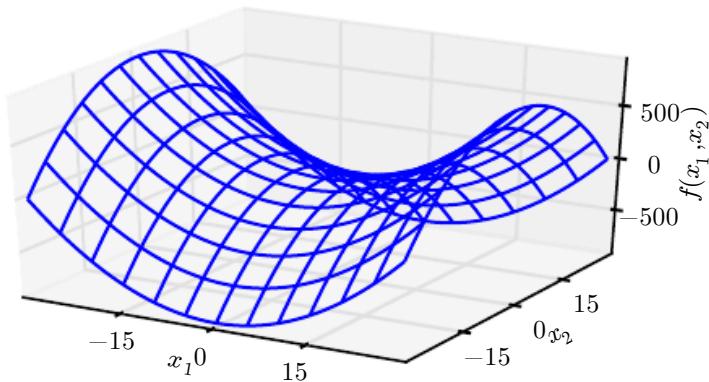


Figure 4.5: A saddle point containing both positive and negative curvature. The function in this example is $f(\mathbf{x}) = x_1^2 - x_2^2$. Along the axis corresponding to x_1 , the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding to x_2 , the function curves downward. This direction is an eigenvector of the Hessian with negative eigenvalue. The name “saddle point” derives from the saddle-like shape of this function. This is the quintessential example of a function with a saddle point. In more than one dimension, it is not necessary to have an eigenvalue of 0 to get a saddle point: it is only necessary to have both positive and negative eigenvalues. We can think of a saddle point with both signs of eigenvalues as being a local maximum within one cross section and a local minimum within another cross section.

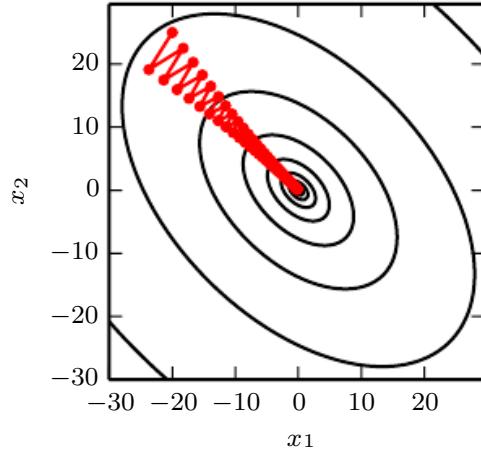


Figure 4.6: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent to minimize a quadratic function $f(\mathbf{x})$ whose Hessian matrix has condition number 5. This means that the direction of most curvature has five times more curvature than the direction of least curvature. In this case, the most curvature is in the direction $[1, 1]^\top$, and the least curvature is in the direction $[1, -1]^\top$. The red lines indicate the path followed by gradient descent. This very elongated quadratic function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls because they are the steepest feature. Since the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration. The large positive eigenvalue of the Hessian corresponding to the eigenvector pointed in this direction indicates that this directional derivative is rapidly increasing, so an optimization algorithm based on the Hessian could predict that the steepest direction is not actually a promising search direction in this context.

This issue can be resolved by using information from the Hessian matrix to guide the search. The simplest method for doing so is known as **Newton's method**. Newton's method is based on using a second-order Taylor series expansion to approximate $f(\mathbf{x})$ near some point $\mathbf{x}^{(0)}$:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H}(f)(\mathbf{x}^{(0)}) (\mathbf{x} - \mathbf{x}^{(0)}). \quad (4.11)$$

If we then solve for the critical point of this function, we obtain

$$\mathbf{x}^* = \mathbf{x}^{(0)} - \mathbf{H}(f)(\mathbf{x}^{(0)})^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}). \quad (4.12)$$

When f is a positive definite quadratic function, Newton's method consists of applying equation 4.12 once to jump to the minimum of the function directly. When f is not truly quadratic but can be locally approximated as a positive definite quadratic, Newton's method consists of applying equation 4.12 multiple times. Iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would. This is a useful property near a local minimum, but it can be a harmful property near a saddle point. As discussed in section 8.2.3, Newton's method is only appropriate when the nearby critical point is a minimum (all the eigenvalues of the Hessian are positive), whereas gradient descent is not attracted to saddle points unless the gradient points toward them.

Optimization algorithms that use only the gradient, such as gradient descent, are called **first-order optimization algorithms**. Optimization algorithms that also use the Hessian matrix, such as Newton's method, are called **second-order optimization algorithms** (Nocedal and Wright, 2006).

The optimization algorithms employed in most contexts in this book are applicable to a wide variety of functions but come with almost no guarantees. Deep learning algorithms tend to lack guarantees because the family of functions used in deep learning is quite complicated. In many other fields, the dominant approach to optimization is to design optimization algorithms for a limited family of functions.

In the context of deep learning, we sometimes gain some guarantees by restricting ourselves to functions that are either **Lipschitz continuous** or have Lipschitz continuous derivatives. A Lipschitz continuous function is a function f whose rate of change is bounded by a **Lipschitz constant** \mathcal{L} :

$$\forall \mathbf{x}, \forall \mathbf{y}, |f(\mathbf{x}) - f(\mathbf{y})| \leq \mathcal{L} \|\mathbf{x} - \mathbf{y}\|_2. \quad (4.13)$$

This property is useful because it enables us to quantify our assumption that a small change in the input made by an algorithm such as gradient descent will have

a small change in the output. Lipschitz continuity is also a fairly weak constraint, and many optimization problems in deep learning can be made Lipschitz continuous with relatively minor modifications.

Perhaps the most successful field of specialized optimization is **convex optimization**. Convex optimization algorithms are able to provide many more guarantees by making stronger restrictions. These algorithms are applicable only to convex functions—functions for which the Hessian is positive semidefinite everywhere. Such functions are well-behaved because they lack saddle points, and all their local minima are necessarily global minima. However, most problems in deep learning are difficult to express in terms of convex optimization. Convex optimization is used only as a subroutine of some deep learning algorithms. Ideas from the analysis of convex optimization algorithms can be useful for proving the convergence of deep learning algorithms, but in general, the importance of convex optimization is greatly diminished in the context of deep learning. For more information about convex optimization, see [Boyd and Vandenberghe \(2004\)](#) or [Rockafellar \(1997\)](#).

4.4 Constrained Optimization

Sometimes we wish not only to maximize or minimize a function $f(\mathbf{x})$ over all possible values of \mathbf{x} . Instead we may wish to find the maximal or minimal value of $f(\mathbf{x})$ for values of \mathbf{x} in some set \mathbb{S} . This is known as **constrained optimization**. Points \mathbf{x} that lie within the set \mathbb{S} are called **feasible** points in constrained optimization terminology.

We often wish to find a solution that is small in some sense. A common approach in such situations is to impose a norm constraint, such as $\|\mathbf{x}\| \leq 1$.

One simple approach to constrained optimization is simply to modify gradient descent taking the constraint into account. If we use a small constant step size ϵ , we can make gradient descent steps, then project the result back into \mathbb{S} . If we use a line search, we can search only over step sizes ϵ that yield new \mathbf{x} points that are feasible, or we can project each point on the line back into the constraint region. When possible, this method can be made more efficient by projecting the gradient into the tangent space of the feasible region before taking the step or beginning the line search ([Rosen, 1960](#)).

A more sophisticated approach is to design a different, unconstrained optimization problem whose solution can be converted into a solution to the original, constrained optimization problem. For example, if we want to minimize $f(\mathbf{x})$ for

$\mathbf{x} \in \mathbb{R}^2$ with \mathbf{x} constrained to have exactly unit L^2 norm, we can instead minimize $g(\theta) = f([\cos \theta, \sin \theta]^\top)$ with respect to θ , then return $[\cos \theta, \sin \theta]$ as the solution to the original problem. This approach requires creativity; the transformation between optimization problems must be designed specifically for each case we encounter.

The **Karush–Kuhn–Tucker** (KKT) approach¹ provides a very general solution to constrained optimization. With the KKT approach, we introduce a new function called the **generalized Lagrangian** or **generalized Lagrange function**.

To define the Lagrangian, we first need to describe \mathbb{S} in terms of equations and inequalities. We want a description of \mathbb{S} in terms of m functions $g^{(i)}$ and n functions $h^{(j)}$ so that $\mathbb{S} = \{\mathbf{x} \mid \forall i, g^{(i)}(\mathbf{x}) = 0 \text{ and } \forall j, h^{(j)}(\mathbf{x}) \leq 0\}$. The equations involving $g^{(i)}$ are called the **equality constraints**, and the inequalities involving $h^{(j)}$ are called **inequality constraints**.

We introduce new variables λ_i and α_j for each constraint, these are called the KKT multipliers. The generalized Lagrangian is then defined as

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) + \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.14)$$

We can now solve a constrained minimization problem using unconstrained optimization of the generalized Lagrangian. As long as at least one feasible point exists and $f(\mathbf{x})$ is not permitted to have value ∞ , then

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \quad (4.15)$$

has the same optimal objective function value and set of optimal points \mathbf{x} as

$$\min_{\mathbf{x} \in \mathbb{S}} f(\mathbf{x}). \quad (4.16)$$

This follows because any time the constraints are satisfied,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}), \quad (4.17)$$

while any time a constraint is violated,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty. \quad (4.18)$$

These properties guarantee that no infeasible point can be optimal, and that the optimum within the feasible points is unchanged.

¹The KKT approach generalizes the method of **Lagrange multipliers**, which allows equality constraints but not inequality constraints.

To perform constrained maximization, we can construct the generalized Lagrange function of $-f(\mathbf{x})$, which leads to this optimization problem:

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} -f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) + \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.19)$$

We may also convert this to a problem with maximization in the outer loop:

$$\max_{\mathbf{x}} \min_{\boldsymbol{\lambda}} \min_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) - \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.20)$$

The sign of the term for the equality constraints does not matter; we may define it with addition or subtraction as we wish, because the optimization is free to choose any sign for each λ_i .

The inequality constraints are particularly interesting. We say that a constraint $h^{(i)}(\mathbf{x})$ is **active** if $h^{(i)}(\mathbf{x}^*) = 0$. If a constraint is not active, then the solution to the problem found using that constraint would remain at least a local solution if that constraint were removed. It is possible that an inactive constraint excludes other solutions. For example, a convex problem with an entire region of globally optimal points (a wide, flat region of equal cost) could have a subset of this region eliminated by constraints, or a nonconvex problem could have better local stationary points excluded by a constraint that is inactive at convergence. Yet the point found at convergence remains a stationary point whether or not the inactive constraints are included. Because an inactive $h^{(i)}$ has negative value, then the solution to $\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ will have $\alpha_i = 0$. We can thus observe that at the solution, $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = \mathbf{0}$. In other words, for all i , we know that at least one of the constraints $\alpha_i \geq 0$ or $h^{(i)}(\mathbf{x}) \leq 0$ must be active at the solution. To gain some intuition for this idea, we can say that either the solution is on the boundary imposed by the inequality and we must use its KKT multiplier to influence the solution to \mathbf{x} , or the inequality has no influence on the solution and we represent this by zeroing out its KKT multiplier.

A simple set of properties describe the optimal points of constrained optimization problems. These properties are called the Karush-Kuhn-Tucker (KKT) conditions ([Karush, 1939](#); [Kuhn and Tucker, 1951](#)). They are necessary conditions, but not always sufficient conditions, for a point to be optimal. The conditions are:

- The gradient of the generalized Lagrangian is zero.
- All constraints on both \mathbf{x} and the KKT multipliers are satisfied.
- The inequality constraints exhibit “complementary slackness”: $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = \mathbf{0}$.

For more information about the KKT approach, see [Nocedal and Wright \(2006\)](#).

4.5 Example: Linear Least Squares

Suppose we want to find the value of \mathbf{x} that minimizes

$$f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2. \quad (4.21)$$

Specialized linear algebra algorithms can solve this problem efficiently; however, we can also explore how to solve it using gradient-based optimization as a simple example of how these techniques work.

First, we need to obtain the gradient:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}^\top (\mathbf{Ax} - \mathbf{b}) = \mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}. \quad (4.22)$$

We can then follow this gradient downhill, taking small steps. See algorithm 4.1 for details.

Algorithm 4.1 An algorithm to minimize $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2$ with respect to \mathbf{x} using gradient descent, starting from an arbitrary value of \mathbf{x} .

Set the step size (ϵ) and tolerance (δ) to small, positive numbers.

```
while \|A^\top Ax - A^\top b\|_2 > \delta do
    x \leftarrow x - \epsilon (A^\top Ax - A^\top b)
end while
```

One can also solve this problem using Newton's method. In this case, because the true function is quadratic, the quadratic approximation employed by Newton's method is exact, and the algorithm converges to the global minimum in a single step.

Now suppose we wish to minimize the same function, but subject to the constraint $\mathbf{x}^\top \mathbf{x} \leq 1$. To do so, we introduce the Lagrangian

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda (\mathbf{x}^\top \mathbf{x} - 1). \quad (4.23)$$

We can now solve the problem

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda). \quad (4.24)$$

The smallest-norm solution to the unconstrained least-squares problem may be found using the Moore-Penrose pseudoinverse: $\mathbf{x} = \mathbf{A}^+ \mathbf{b}$. If this point is feasible, then it is the solution to the constrained problem. Otherwise, we must find a

solution where the constraint is active. By differentiating the Lagrangian with respect to \mathbf{x} , we obtain the equation

$$\mathbf{A}^\top \mathbf{A}\mathbf{x} - \mathbf{A}^\top \mathbf{b} + 2\lambda\mathbf{x} = 0. \quad (4.25)$$

This tells us that the solution will take the form

$$\mathbf{x} = (\mathbf{A}^\top \mathbf{A} + 2\lambda\mathbf{I})^{-1} \mathbf{A}^\top \mathbf{b}. \quad (4.26)$$

The magnitude of λ must be chosen such that the result obeys the constraint. We can find this value by performing gradient ascent on λ . To do so, observe

$$\frac{\partial}{\partial \lambda} L(\mathbf{x}, \lambda) = \mathbf{x}^\top \mathbf{x} - 1. \quad (4.27)$$

When the norm of \mathbf{x} exceeds 1, this derivative is positive, so to follow the derivative uphill and increase the Lagrangian with respect to λ , we increase λ . Because the coefficient on the $\mathbf{x}^\top \mathbf{x}$ penalty has increased, solving the linear equation for \mathbf{x} will now yield a solution with a smaller norm. The process of solving the linear equation and adjusting λ continues until \mathbf{x} has the correct norm and the derivative on λ is 0.

This concludes the mathematical preliminaries that we use to develop machine learning algorithms. We are now ready to build and analyze some full-fledged learning systems.

Chapter 5

Machine Learning Basics

Deep learning is a specific kind of machine learning. To understand deep learning well, one must have a solid understanding of the basic principles of machine learning. This chapter provides a brief course in the most important general principles that are applied throughout the rest of the book. Novice readers or those who want a wider perspective are encouraged to consider machine learning textbooks with a more comprehensive coverage of the fundamentals, such as Murphy (2012) or Bishop (2006). If you are already familiar with machine learning basics, feel free to skip ahead to section 5.11. That section covers some perspectives on traditional machine learning techniques that have strongly influenced the development of deep learning algorithms.

We begin with a definition of what a learning algorithm is and present an example: the linear regression algorithm. We then proceed to describe how the challenge of fitting the training data differs from the challenge of finding patterns that generalize to new data. Most machine learning algorithms have settings called *hyperparameters*, which must be determined outside the learning algorithm itself; we discuss how to set these using additional data. Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions; we therefore present the two central approaches to statistics: frequentist estimators and Bayesian inference. Most machine learning algorithms can be divided into the categories of supervised learning and unsupervised learning; we describe these categories and give some examples of simple learning algorithms from each category. Most deep learning algorithms are based on an optimization algorithm called stochastic gradient

descent. We describe how to combine various algorithm components, such as an optimization algorithm, a cost function, a model, and a dataset, to build a machine learning algorithm. Finally, in section 5.11, we describe some of the factors that have limited the ability of traditional machine learning to generalize. These challenges have motivated the development of deep learning algorithms that overcome these obstacles.

5.1 Learning Algorithms

A machine learning algorithm is an algorithm that is able to learn from data. But what do we mean by learning? Mitchell (1997) provides a succinct definition: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .” One can imagine a wide variety of experiences E , tasks T , and performance measures P , and we do not attempt in this book to formally define what may be used for each of these entities. Instead, in the following sections, we provide intuitive descriptions and examples of the different kinds of tasks, performance measures, and experiences that can be used to construct machine learning algorithms.

5.1.1 The Task, T

Machine learning enables us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientific and philosophical point of view, machine learning is interesting because developing our understanding of it entails developing our understanding of the principles that underlie intelligence.

In this relatively formal definition of the word “task,” the process of learning itself is not the task. Learning is our means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually.

Machine learning tasks are usually described in terms of how the machine learning system should process an **example**. An example is a collection of **features** that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $\mathbf{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature. For example, the features of an image are usually the values of the pixels in the image.

Many kinds of tasks can be solved with machine learning. Some of the most common machine learning tasks include the following:

- **Classification:** In this type of task, the computer program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$. When $y = f(\mathbf{x})$, the model assigns an input described by vector \mathbf{x} to a category identified by numeric code y . There are other variants of the classification task, for example, where f outputs a probability distribution over classes. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image. For example, the Willow Garage PR2 robot is able to act as a waiter that can recognize different kinds of drinks and deliver them to people on command (Goodfellow *et al.*, 2010). Modern object recognition is best accomplished with deep learning (Krizhevsky *et al.*, 2012; Ioffe and Szegedy, 2015). Object recognition is the same basic technology that enables computers to recognize faces (Taigman *et al.*, 2014), which can be used to automatically tag people in photo collections and for computers to interact more naturally with their users.
- **Classification with missing inputs:** Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. To solve the classification task, the learning algorithm only has to define a *single* function mapping from a vector input to a categorical output. When some of the inputs may be missing, rather than providing a single classification function, the learning algorithm must learn a *set* of functions. Each function corresponds to classifying \mathbf{x} with a different subset of its inputs missing. This kind of situation arises frequently in medical diagnosis, because many kinds of medical tests are expensive or invasive. One way to efficiently define such a large set of functions is to learn a probability distribution over all the relevant variables, then solve the classification task by marginalizing out the missing variables. With n input variables, we can now obtain all 2^n different classification functions needed for each possible set of missing inputs, but the computer program needs to learn only a single function describing the joint probability distribution. See Goodfellow *et al.* (2013b) for an example of a deep probabilistic model applied to such a task in this way. Many of the other tasks described in this section can also be generalized to work with missing inputs; classification with missing inputs is just one example of what machine learning can do.

- **Regression:** In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums), or the prediction of future prices of securities. These kinds of predictions are also used for algorithmic trading.
- **Transcription:** In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe the information into discrete textual form. For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text in the form of a sequence of characters (e.g., in ASCII or Unicode format). Google Street View uses deep learning to process address numbers in this way ([Goodfellow et al., 2014d](#)). Another example is speech recognition, where the computer program is provided an audio waveform and emits a sequence of characters or word ID codes describing the words that were spoken in the audio recording. Deep learning is a crucial component of modern speech recognition systems used at major companies, including Microsoft, IBM and Google ([Hinton et al., 2012b](#)).
- **Machine translation:** In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. This is commonly applied to natural languages, such as translating from English to French. Deep learning has recently begun to have an important impact on this kind of task ([Sutskever et al., 2014](#); [Bahdanau et al., 2015](#)).
- **Structured output:** Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements. This is a broad category and subsumes the transcription and translation tasks described above, as well as many other tasks. One example is parsing—mapping a natural language sentence into a tree that describes its grammatical structure by tagging nodes of the trees as being verbs, nouns, adverbs, and so on. See [Collobert \(2011\)](#) for an example of deep learning applied to a parsing task. Another example is pixel-wise segmentation of images, where the computer program assigns every pixel in an image to a specific category.

For example, deep learning can be used to annotate the locations of roads in aerial photographs (Mnih and Hinton, 2010). The output form need not mirror the structure of the input as closely as in these annotation-style tasks. For example, in image captioning, the computer program observes an image and outputs a natural language sentence describing the image (Kiros *et al.*, 2014a,b; Mao *et al.*, 2015; Vinyals *et al.*, 2015b; Donahue *et al.*, 2014; Karpathy and Li, 2015; Fang *et al.*, 2015; Xu *et al.*, 2015). These tasks are called *structured output tasks* because the program must output several values that are all tightly interrelated. For example, the words produced by an image captioning program must form a valid sentence.

- **Anomaly detection:** In this type of task, the computer program sifts through a set of events or objects and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection. By modeling your purchasing habits, a credit card company can detect misuse of your cards. If a thief steals your credit card or credit card information, the thief’s purchases will often come from a different probability distribution over purchase types than your own. The credit card company can prevent fraud by placing a hold on an account as soon as that card has been used for an uncharacteristic purchase. See Chandola *et al.* (2009) for a survey of anomaly detection methods.
- **Synthesis and sampling:** In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. Synthesis and sampling via machine learning can be useful for media applications when generating large volumes of content by hand would be expensive, boring, or require too much time. For example, video games can automatically generate textures for large objects or landscapes, rather than requiring an artist to manually label each pixel (Luo *et al.*, 2013). In some cases, we want the sampling or synthesis procedure to generate a specific kind of output given the input. For example, in a speech synthesis task, we provide a written sentence and ask the program to emit an audio waveform containing a spoken version of that sentence. This is a kind of structured output task, but with the added qualification that there is no single correct output for each input, and we explicitly desire a large amount of variation in the output, in order for the output to seem more natural and realistic.
- **Imputation of missing values:** In this type of task, the machine learning algorithm is given a new example $\mathbf{x} \in \mathbb{R}^n$, but with some entries x_i of \mathbf{x}

missing. The algorithm must provide a prediction of the values of the missing entries.

- **Denoising:** In this type of task, the machine learning algorithm is given as input a *corrupted example* $\tilde{\mathbf{x}} \in \mathbb{R}^n$ obtained by an unknown corruption process from a *clean example* $\mathbf{x} \in \mathbb{R}^n$. The learner must predict the clean example \mathbf{x} from its corrupted version $\tilde{\mathbf{x}}$, or more generally predict the conditional probability distribution $p(\mathbf{x} | \tilde{\mathbf{x}})$.
- **Density estimation or probability mass function estimation:** In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{\text{model}}(\mathbf{x})$ can be interpreted as a probability density function (if \mathbf{x} is continuous) or a probability mass function (if \mathbf{x} is discrete) on the space that the examples were drawn from. To do such a task well (we will specify exactly what that means when we discuss performance measures P), the algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur. Most of the tasks described above require the learning algorithm to at least implicitly capture the structure of the probability distribution. Density estimation enables us to explicitly capture that distribution. In principle, we can then perform computations on that distribution to solve the other tasks as well. For example, if we have performed density estimation to obtain a probability distribution $p(\mathbf{x})$, we can use that distribution to solve the missing value imputation task. If a value x_i is missing, and all the other values, denoted \mathbf{x}_{-i} , are given, then we know the distribution over it is given by $p(x_i | \mathbf{x}_{-i})$. In practice, density estimation does not always enable us to solve all these related tasks, because in many cases the required operations on $p(\mathbf{x})$ are computationally intractable.

Of course, many other tasks and types of tasks are possible. The types of tasks we list here are intended only to provide examples of what machine learning can do, not to define a rigid taxonomy of tasks.

5.1.2 The Performance Measure, P

To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task T being carried out by the system.

For tasks such as classification, classification with missing inputs, and transcription, we often measure the **accuracy** of the model. Accuracy is just the

proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the **error rate**, the proportion of examples for which the model produces an incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not. For tasks such as density estimation, it does not make sense to measure accuracy, error rate, or any other kind of 0-1 loss. Instead, we must use a different performance metric that gives the model a continuous-valued score for each example. The most common approach is to report the average log-probability the model assigns to some examples.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a **test set** of data that is separate from the data used for training the machine learning system.

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system.

In some cases, this is because it is difficult to decide what should be measured. For example, when performing a transcription task, should we measure the accuracy of the system at transcribing entire sequences, or should we use a more fine-grained performance measure that gives partial credit for getting some elements of the sequence correct? When performing a regression task, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes? These kinds of design choices depend on the application.

In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. For example, this arises frequently in the context of density estimation. Many of the best probabilistic models represent probability distributions only implicitly. Computing the actual probability value assigned to a specific point in space in many such models is intractable. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion.

5.1.3 The Experience, E

Machine learning algorithms can be broadly categorized as **unsupervised** or **supervised** by what kind of experience they are allowed to have during the learning process.

Most of the learning algorithms in this book can be understood as being allowed

to experience an entire **dataset**. A dataset is a collection of many examples, as defined in section 5.1.1. Sometimes we call examples **data points**.

One of the oldest datasets studied by statisticians and machine learning researchers is the Iris dataset (Fisher, 1936). It is a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one example. The features within each example are the measurements of each part of the plant: the sepal length, sepal width, petal length and petal width. The dataset also records which species each plant belonged to. Three different species are represented in the dataset.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly, as in density estimation, or implicitly, for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a **label** or **target**. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.

Roughly speaking, unsupervised learning involves observing several examples of a random vector \mathbf{x} and attempting to implicitly or explicitly learn the probability distribution $p(\mathbf{x})$, or some interesting properties of that distribution; while supervised learning involves observing several examples of a random vector \mathbf{x} and an associated value or vector \mathbf{y} , then learning to predict \mathbf{y} from \mathbf{x} , usually by estimating $p(\mathbf{y} \mid \mathbf{x})$. The term **supervised learning** originates from the view of the target \mathbf{y} being provided by an instructor or teacher who shows the machine learning system what to do. In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

Unsupervised learning and supervised learning are not formally defined terms. The lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $\mathbf{x} \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(\mathbf{x}) = \prod_{i=1}^n p(x_i \mid x_1, \dots, x_{i-1}). \quad (5.1)$$

This decomposition means that we can solve the ostensibly unsupervised problem of

modeling $p(\mathbf{x})$ by splitting it into n supervised learning problems. Alternatively, we can solve the supervised learning problem of learning $p(y | \mathbf{x})$ by using traditional unsupervised learning technologies to learn the joint distribution $p(\mathbf{x}, y)$, then inferring

$$p(y | \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_y p(\mathbf{x}, y)}. \quad (5.2)$$

Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

Other variants of the learning paradigm are possible. For example, in semi-supervised learning, some examples include a supervision target but others do not. In multi-instance learning, an entire collection of examples is labeled as containing or not containing an example of a class, but the individual members of the collection are not labeled. For a recent example of multi-instance learning with deep models, see [Kotzias *et al.* \(2015\)](#).

Some machine learning algorithms do not just experience a fixed dataset. For example, **reinforcement learning** algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences. Such algorithms are beyond the scope of this book. Please see [Sutton and Barto \(1998\)](#) or [Bertsekas and Tsitsiklis \(1996\)](#) for information about reinforcement learning, and [Mnih *et al.* \(2013\)](#) for the deep learning approach to reinforcement learning.

Most machine learning algorithms simply experience a dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples, which are in turn collections of features.

One common way of describing a dataset is with a **design matrix**. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\mathbf{X} \in \mathbb{R}^{150 \times 4}$, where $X_{i,1}$ is the sepal length of plant i , $X_{i,2}$ is the sepal width of plant i , etc. We describe most of the learning algorithms in this book in terms of how they operate on design matrix datasets.

Of course, to describe a dataset as a design matrix, it must be possible to describe each example as a vector, and each of these vectors must be the same size. This is not always possible. For example, if you have a collection of photographs with different widths and heights, then different photographs will contain different numbers of pixels, so not all the photographs may be described with the same

length of vector. In Section 9.7 and chapter 10, we describe how to handle different types of such heterogeneous data. In cases like these, rather than describing the dataset as a matrix with m rows, we describe it as a set containing m elements: $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\}$. This notation does not imply that any two example vectors $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ have the same size.

In the case of supervised learning, the example contains a label or target as well as a collection of features. For example, if we want to use a learning algorithm to perform object recognition from photographs, we need to specify which object appears in each of the photos. We might do this with a numeric code, with 0 signifying a person, 1 signifying a car, 2 signifying a cat, and so forth. Often when working with a dataset containing a design matrix of feature observations \mathbf{X} , we also provide a vector of labels \mathbf{y} , with y_i providing the label for example i .

Of course, sometimes the label may be more than just a single number. For example, if we want to train a speech recognition system to transcribe entire sentences, then the label for each example sentence is a sequence of words.

Just as there is no formal definition of supervised and unsupervised learning, there is no rigid taxonomy of datasets or experiences. The structures described here cover most cases, but it is always possible to design new ones for new applications.

5.1.4 Example: Linear Regression

Our definition of a machine learning algorithm as an algorithm that is capable of improving a computer program's performance at some task via experience is somewhat abstract. To make this more concrete, we present an example of a simple machine learning algorithm: **linear regression**. We will return to this example repeatedly as we introduce more machine learning concepts that help to understand the algorithm's behavior.

As the name implies, linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $\mathbf{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. The output of linear regression is a linear function of the input. Let \hat{y} be the value that our model predicts y should take on. We define the output to be

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \quad (5.3)$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of **parameters**.

Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features. We can think of \mathbf{w} as a set of **weights** that determine how

each feature affects the prediction. If a feature x_i receives a positive weight w_i , then increasing the value of that feature increases the value of our prediction \hat{y} . If a feature receives a negative weight, then increasing the value of that feature decreases the value of our prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction.

We thus have a definition of our task T : to predict y from \mathbf{x} by outputting $\hat{y} = \mathbf{w}^\top \mathbf{x}$. Next we need a definition of our performance measure, P .

Suppose that we have a design matrix of m example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of y for each of these examples. Because this dataset will only be used for evaluation, we call it the test set. We refer to the design matrix of inputs as $\mathbf{X}^{(\text{test})}$ and the vector of regression targets as $\mathbf{y}^{(\text{test})}$.

One way of measuring the performance of the model is to compute the **mean squared error** of the model on the test set. If $\hat{\mathbf{y}}^{(\text{test})}$ gives the predictions of the model on the test set, then the mean squared error is given by

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i (\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})})_i^2. \quad (5.4)$$

Intuitively, one can see that this error measure decreases to 0 when $\hat{\mathbf{y}}^{(\text{test})} = \mathbf{y}^{(\text{test})}$. We can also see that

$$\text{MSE}_{\text{test}} = \frac{1}{m} \|\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})}\|_2^2, \quad (5.5)$$

so the error increases whenever the Euclidean distance between the predictions and the targets increases.

To make a machine learning algorithm, we need to design an algorithm that will improve the weights \mathbf{w} in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$. One intuitive way of doing this (which we justify later, in section 5.5.1) is just to minimize the mean squared error on the training set, $\text{MSE}_{\text{train}}$.

To minimize $\text{MSE}_{\text{train}}$, we can simply solve for where its gradient is **0**:

$$\nabla_{\mathbf{w}} \text{MSE}_{\text{train}} = 0 \quad (5.6)$$

$$\Rightarrow \nabla_{\mathbf{w}} \frac{1}{m} \|\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\|_2^2 = 0 \quad (5.7)$$

$$\Rightarrow \frac{1}{m} \nabla_{\mathbf{w}} \|\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2 = 0 \quad (5.8)$$

$$\Rightarrow \nabla_{\mathbf{w}} (\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})})^\top (\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})}) = 0 \quad (5.9)$$

$$\Rightarrow \nabla_{\mathbf{w}} (\mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} + \mathbf{y}^{(\text{train})\top} \mathbf{y}^{(\text{train})}) = 0 \quad (5.10)$$

$$\Rightarrow 2\mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2\mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} = 0 \quad (5.11)$$

$$\Rightarrow \mathbf{w} = (\mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})})^{-1} \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} \quad (5.12)$$

The system of equations whose solution is given by equation 5.12 is known as the **normal equations**. Evaluating equation 5.12 constitutes a simple learning algorithm. For an example of the linear regression learning algorithm in action, see figure 5.1.

It is worth noting that the term **linear regression** is often used to refer to a slightly more sophisticated model with one additional parameter—an intercept term b . In this model

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b, \quad (5.13)$$

so the mapping from parameters to predictions is still a linear function but the mapping from features to predictions is now an affine function. This extension to

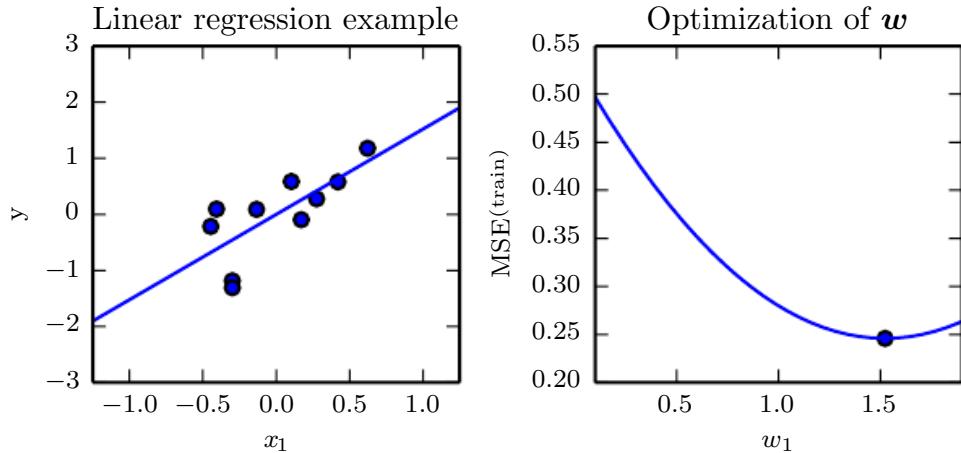


Figure 5.1: A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector \mathbf{w} contains only a single parameter to learn, w_1 . (*Left*) Observe that linear regression learns to set w_1 such that the line $y = w_1 x$ comes as close as possible to passing through all the training points. (*Right*) The plotted point indicates the value of w_1 found by the normal equations, which we can see minimizes the mean squared error on the training set.

affine functions means that the plot of the model’s predictions still looks like a line, but it need not pass through the origin. Instead of adding the bias parameter b , one can continue to use the model with only weights but augment \mathbf{x} with an extra entry that is always set to 1. The weight corresponding to the extra 1 entry plays the role of the bias parameter. We frequently use the term “linear” when referring to affine functions throughout this book.

The intercept term b is often called the **bias** parameter of the affine transformation. This terminology derives from the point of view that the output of the transformation is biased toward being b in the absence of any input. This term is different from the idea of a statistical bias, in which a statistical estimation algorithm’s expected estimate of a quantity is not equal to the true quantity.

Linear regression is of course an extremely simple and limited learning algorithm, but it provides an example of how a learning algorithm can work. In subsequent sections we describe some of the basic principles underlying learning algorithm design and demonstrate how these principles can be used to build more complicated learning algorithms.

5.2 Capacity, Overfitting and Underfitting

The central challenge in machine learning is that our algorithm must perform well on *new, previously unseen* inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization**.

Typically, when training a machine learning model, we have access to a training set; we can compute some error measure on the training set, called the **training error**; and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the **generalization error**, also called the **test error**, to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a **test set** of examples that were collected separately from the training set.

In our linear regression example, we trained the model by minimizing the training error,

$$\frac{1}{m^{(\text{train})}} \|\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2, \quad (5.14)$$

but we actually care about the test error, $\frac{1}{m^{(\text{test})}} \|\mathbf{X}^{(\text{test})}\mathbf{w} - \mathbf{y}^{(\text{test})}\|_2^2$.

How can we affect performance on the test set when we can observe only the training set? The field of **statistical learning theory** provides some answers. If the training and the test set are collected arbitrarily, there is indeed little we can do. If we are allowed to make some assumptions about how the training and test set are collected, then we can make some progress.

The training and test data are generated by a probability distribution over datasets called the **data-generating process**. We typically make a set of assumptions known collectively as the **i.i.d. assumptions**. These assumptions are that the examples in each dataset are **independent** from each other, and that the training set and test set are **identically distributed**, drawn from the same probability distribution as each other. This assumption enables us to describe the data-generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example. We call that shared underlying distribution the **data-generating distribution**, denoted p_{data} . This probabilistic framework and the i.i.d. assumptions enables us to mathematically study the relationship between training error and test error.

One immediate connection we can observe between training error and test error is that the expected training error of a randomly selected model is equal to the expected test error of that model. Suppose we have a probability distribution $p(\mathbf{x}, y)$ and we sample from it repeatedly to generate the training set and the test set. For some fixed value \mathbf{w} , the expected training set error is exactly the same as the expected test set error, because both expectations are formed using the same dataset sampling process. The only difference between the two conditions is the name we assign to the dataset we sample.

Of course, when we use a machine learning algorithm, we do not fix the parameters ahead of time, then sample both datasets. We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform are its ability to

1. Make the training error small.
2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: **underfitting** and **overfitting**. Underfitting occurs when the model is not able to

obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its **capacity**. Informally, a model’s capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its **hypothesis space**, the set of functions that the learning algorithm is allowed to select as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model’s capacity.

A polynomial of degree 1 gives us the linear regression model with which we are already familiar, with the prediction

$$\hat{y} = b + wx. \quad (5.15)$$

By introducing x^2 as another feature provided to the linear regression model, we can learn a model that is quadratic as a function of x :

$$\hat{y} = b + w_1x + w_2x^2. \quad (5.16)$$

Though this model implements a quadratic function of its *input*, the output is still a linear function of the *parameters*, so we can still use the normal equations to train the model in closed form. We can continue to add more powers of x as additional features, for example, to obtain a polynomial of degree 9:

$$\hat{y} = b + \sum_{i=1}^9 w_i x^i. \quad (5.17)$$

Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit.

Figure 5.2 shows this principle in action. We compare a linear, quadratic and degree-9 predictor attempting to fit a problem where the true underlying

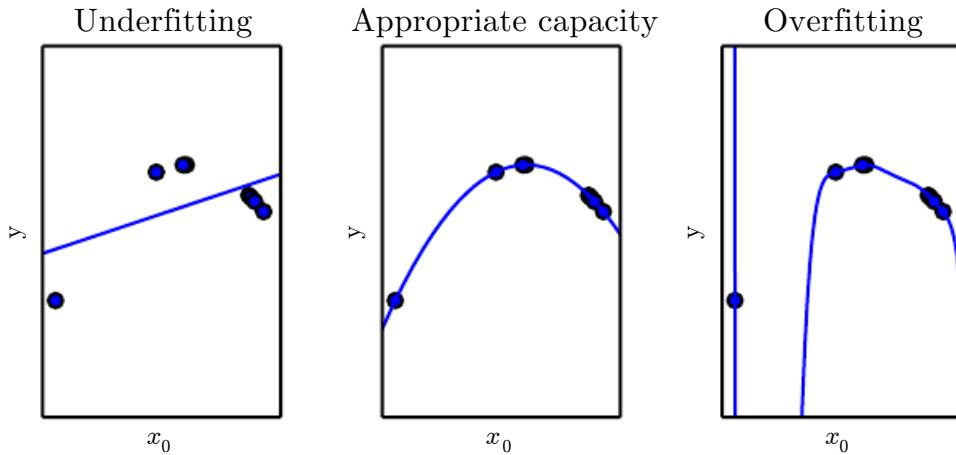


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. (*Left*) A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. (*Center*) A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. (*Right*) A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudoinverse to solve the underdetermined normal equations. The solution passes through all the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

function is quadratic. The linear function is unable to capture the curvature in the true underlying problem, so it underfits. The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely many other functions that pass exactly through the training points, because we have more parameters than training examples. We have little chance of choosing a solution that generalizes well when so many wildly different solutions exist. In this example, the quadratic model is perfectly matched to the true structure of the task, so it generalizes well to new data.

So far we have described only one way of changing a model’s capacity: by changing the number of input features it has, and simultaneously adding new parameters associated with those features. There are in fact many ways to change a model’s capacity. Capacity is not determined only by the choice of model. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective. This is called the **representational capacity** of the model. In many cases, finding the best

function within this family is a difficult optimization problem. In practice, the learning algorithm does not actually find the best function, but merely one that significantly reduces the training error. These additional limitations, such as the imperfection of the optimization algorithm, mean that the learning algorithm's **effective capacity** may be less than the representational capacity of the model family.

Our modern ideas about improving the generalization of machine learning models are refinements of thought dating back to philosophers at least as early as Ptolemy. Many early scholars invoke a principle of parsimony that is now most widely known as **Occam's razor** (c. 1287–1347). This principle states that among competing hypotheses that explain known observations equally well, we should choose the “simplest” one. This idea was formalized and made more precise in the twentieth century by the founders of statistical learning theory (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995).

Statistical learning theory provides various means of quantifying model capacity. Among these, the most well known is the **Vapnik-Chervonenkis dimension**, or VC dimension. The VC dimension measures the capacity of a binary classifier. The VC dimension is defined as being the largest possible value of m for which there exists a training set of m different \mathbf{x} points that the classifier can label arbitrarily.

Quantifying the capacity of the model enables statistical learning theory to make quantitative predictions. The most important results in statistical learning theory show that the discrepancy between training error and generalization error is bounded from above by a quantity that grows as the model capacity grows but shrinks as the number of training examples increases (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). These bounds provide intellectual justification that machine learning algorithms can work, but they are rarely used in practice when working with deep learning algorithms. This is in part because the bounds are often quite loose and in part because it can be quite difficult to determine the capacity of deep learning algorithms. The problem of determining the capacity of a deep learning model is especially difficult because the effective capacity is limited by the capabilities of the optimization algorithm, and we have little theoretical understanding of the general nonconvex optimization problems involved in deep learning.

We must remember that while simpler functions are more likely to generalize (to have a small gap between training and test error), we must still choose a sufficiently complex hypothesis to achieve low training error. Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming the error measure has a minimum value). Typically,

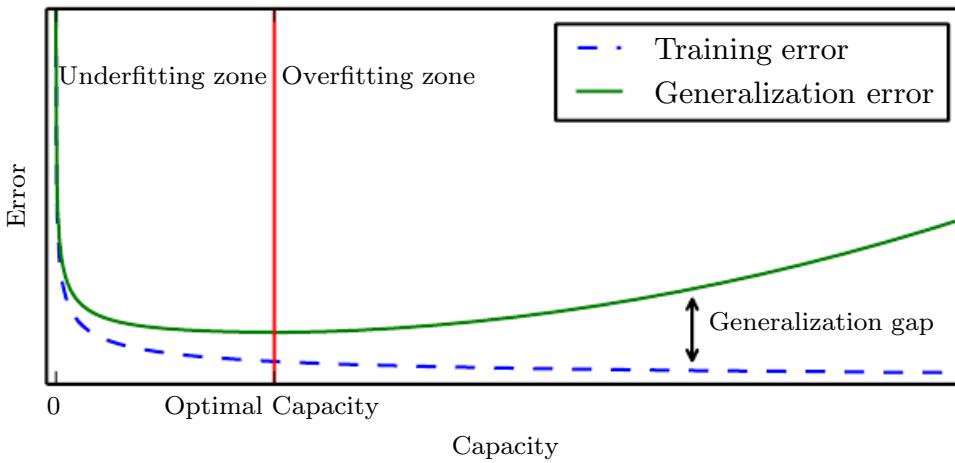


Figure 5.3: Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the **overfitting regime**, where capacity is too large, above the **optimal capacity**.

generalization error has a U-shaped curve as a function of model capacity. This is illustrated in figure 5.3.

To reach the most extreme case of arbitrarily high capacity, we introduce the concept of **nonparametric models**. So far, we have seen only parametric models, such as linear regression. Parametric models learn a function described by a parameter vector whose size is finite and fixed before any data is observed. Nonparametric models have no such limitation.

Sometimes, nonparametric models are just theoretical abstractions (such as an algorithm that searches over all possible probability distributions) that cannot be implemented in practice. However, we can also design practical nonparametric models by making their complexity a function of the training set size. One example of such an algorithm is **nearest neighbor regression**. Unlike linear regression, which has a fixed-length vector of weights, the nearest neighbor regression model simply stores the \mathbf{X} and \mathbf{y} from the training set. When asked to classify a test point \mathbf{x} , the model looks up the nearest entry in the training set and returns the associated regression target. In other words, $\hat{y} = y_i$ where $i = \arg \min \|\mathbf{X}_{i,:} - \mathbf{x}\|_2^2$. The algorithm can also be generalized to distance metrics other than the L^2 norm, such as learned distance metrics (Goldberger *et al.*, 2005). If the algorithm is allowed to break ties by averaging the y_i values for all $\mathbf{X}_{i,:}$ that are tied for nearest, then this algorithm is able to achieve the minimum possible training error (which

might be greater than zero, if two identical inputs are associated with different outputs) on any regression dataset.

Finally, we can also create a nonparametric learning algorithm by wrapping a parametric learning algorithm inside another algorithm that increases the number of parameters as needed. For example, we could imagine an outer loop of learning that changes the degree of the polynomial learned by linear regression on top of a polynomial expansion of the input.

The ideal model is an oracle that simply knows the true probability distribution that generates the data. Even such a model will still incur some error on many problems, because there may still be some noise in the distribution. In the case of supervised learning, the mapping from \mathbf{x} to y may be inherently stochastic, or y may be a deterministic function that involves other variables besides those included in \mathbf{x} . The error incurred by an oracle making predictions from the true distribution $p(\mathbf{x}, y)$ is called the **Bayes error**.

Training and generalization error vary as the size of the training set varies. Expected generalization error can never increase as the number of training examples increases. For nonparametric models, more data yield better generalization until the best possible error is achieved. Any fixed parametric model with less than optimal capacity will asymptote to an error value that exceeds the Bayes error. See figure 5.4 for an illustration. Note that it is possible for the model to have optimal capacity and yet still have a large gap between training and generalization errors. In this situation, we may be able to reduce this gap by gathering more training examples.

5.2.1 The No Free Lunch Theorem

Learning theory claims that a machine learning algorithm can generalize well from a finite training set of examples. This seems to contradict some basic principles of logic. Inductive reasoning, or inferring general rules from a limited set of examples, is not logically valid. To logically infer a rule describing every member of a set, one must have information about every member of that set.

In part, machine learning avoids this problem by offering only probabilistic rules, rather than the entirely certain rules used in purely logical reasoning. Machine learning promises to find rules that are *probably* correct about *most* members of the set they concern.

Unfortunately, even this does not resolve the entire problem. The **no free lunch theorem** for machine learning (Wolpert, 1996) states that, averaged over all possible data-generating distributions, every classification algorithm has the

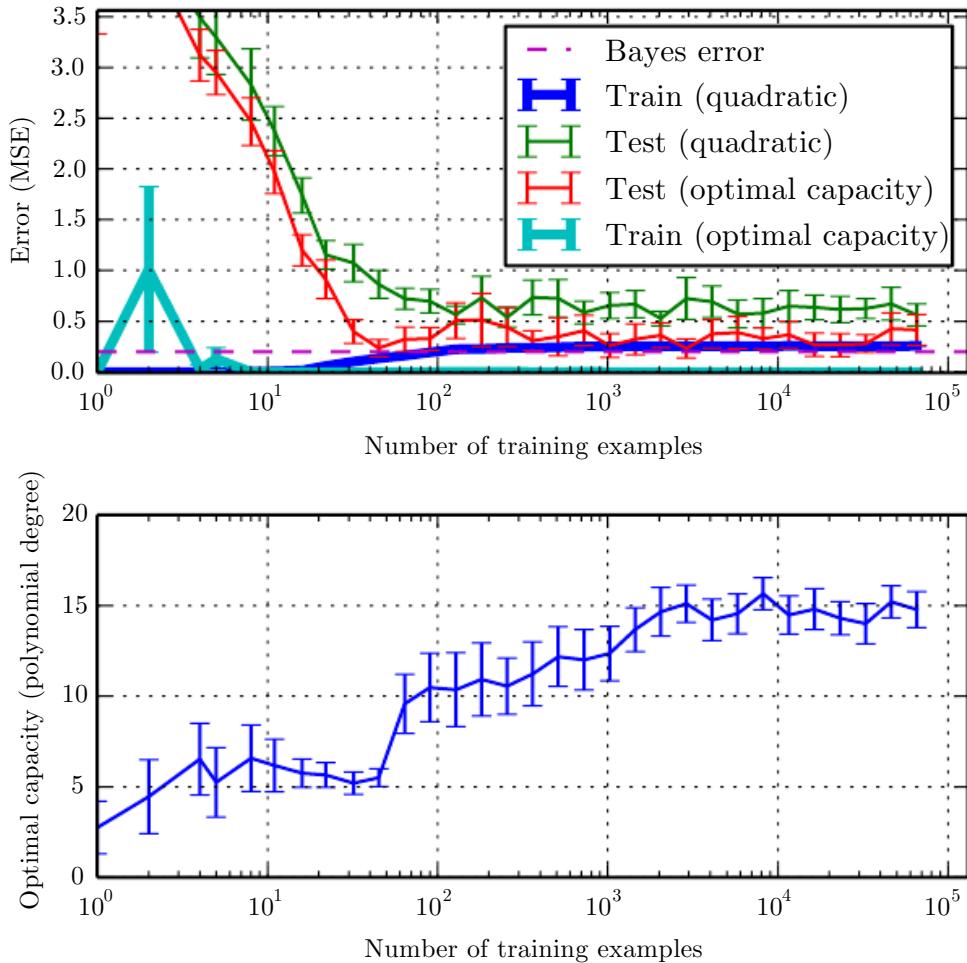


Figure 5.4: The effect of the training dataset size on the train and test error, as well as on the optimal model capacity. We constructed a synthetic regression problem based on adding a moderate amount of noise to a degree-5 polynomial, generated a single test set, and then generated several different sizes of training set. For each size, we generated 40 different training sets in order to plot error bars showing 95 percent confidence intervals. (*Top*)The MSE on the training and test set for two different models: a quadratic model, and a model with degree chosen to minimize the test error. Both are fit in closed form. For the quadratic model, the training error increases as the size of the training set increases. This is because larger datasets are harder to fit. Simultaneously, the test error decreases, because fewer incorrect hypotheses are consistent with the training data. The quadratic model does not have enough capacity to solve the task, so its test error asymptotes to a high value. The test error at optimal capacity asymptotes to the Bayes error. The training error can fall below the Bayes error, due to the ability of the training algorithm to memorize specific instances of the training set. As the training size increases to infinity, the training error of any fixed-capacity model (here, the quadratic model) must rise to at least the Bayes error. (*Bottom*)As the training set size increases, the optimal capacity (shown here as the degree of the optimal polynomial regressor) increases. The optimal capacity plateaus after reaching sufficient complexity to solve the task.

same error rate when classifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally any better than any other. The most sophisticated algorithm we can conceive of has the same average performance (over all possible tasks) as merely predicting that every point belongs to the same class.

Fortunately, these results hold only when we average over *all* possible data-generating distributions. If we make assumptions about the kinds of probability distributions we encounter in real-world applications, then we can design learning algorithms that perform well on these distributions.

This means that the goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, our goal is to understand what kinds of distributions are relevant to the “real world” that an AI agent experiences, and what kinds of machine learning algorithms perform well on data drawn from the kinds of data-generating distributions we care about.

5.2.2 Regularization

The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems that we ask the algorithm to solve, it performs better.

So far, the only method of modifying a learning algorithm that we have discussed concretely is to increase or decrease the model’s representational capacity by adding or removing functions from the hypothesis space of solutions the learning algorithm is able to choose from. We gave the specific example of increasing or decreasing the degree of a polynomial for a regression problem. The view we have described so far is oversimplified.

The behavior of our algorithm is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. The learning algorithm we have studied so far, linear regression, has a hypothesis space consisting of the set of linear functions of its input. These linear functions can be useful for problems where the relationship between inputs and outputs truly is close to linear. They are less useful for problems that behave in a very nonlinear fashion. For example, linear regression would not perform well if we tried to use it to predict $\sin(x)$ from x . We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

We can also give a learning algorithm a preference for one solution over another

in its hypothesis space. This means that both functions are eligible, but one is preferred. The unpreferred solution will be chosen only if it fits the training data significantly better than the preferred solution.

For example, we can modify the training criterion for linear regression to include **weight decay**. To perform linear regression with weight decay, we minimize a sum $J(\mathbf{w})$ comprising both the mean squared error on the training and a criterion that expresses a preference for the weights to have smaller squared L^2 norm. Specifically,

$$J(\mathbf{w}) = \text{MSE}_{\text{train}} + \lambda \mathbf{w}^\top \mathbf{w}, \quad (5.18)$$

where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights. When $\lambda = 0$, we impose no preference, and larger λ forces the weights to become smaller. Minimizing $J(\mathbf{w})$ results in a choice of weights that make a tradeoff between fitting the training data and being small. This gives us solutions that have a smaller slope, or that put weight on fewer of the features. As an example of how we can control a model's tendency to overfit or underfit via weight decay, we can train a high-degree polynomial regression model with different values of λ . See figure 5.5 for the results.

More generally, we can regularize a model that learns a function $f(\mathbf{x}; \boldsymbol{\theta})$ by adding a penalty called a **regularizer** to the cost function. In the case of weight decay, the regularizer is $\Omega(\mathbf{w}) = \mathbf{w}^\top \mathbf{w}$. In chapter 7, we will see that many other regularizers are possible.

Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function.

In our weight decay example, we expressed our preference for linear functions defined with smaller weights explicitly, via an extra term in the criterion we minimize. There are many other ways of expressing preferences for different solutions, both implicitly and explicitly. Together, these different approaches are known as **regularization**. *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.* Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

The no free lunch theorem has made it clear that there is no best machine learning algorithm, and, in particular, no best form of regularization. Instead we must choose a form of regularization that is well suited to the particular task we want to solve. The philosophy of deep learning in general and this book in particular is that a wide range of tasks (such as all the intellectual tasks that

people can do) may all be solved effectively using very general-purpose forms of regularization.

5.3 Hyperparameters and Validation Sets

Most machine learning algorithms have hyperparameters, settings that we can use to control the algorithm's behavior. The values of hyperparameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure in which one learning algorithm learns the best hyperparameters for another learning algorithm).

The polynomial regression example in figure 5.2 has a single hyperparameter: the degree of the polynomial, which acts as a **capacity** hyperparameter. The λ value used to control the strength of weight decay is another example of a hyperparameter.

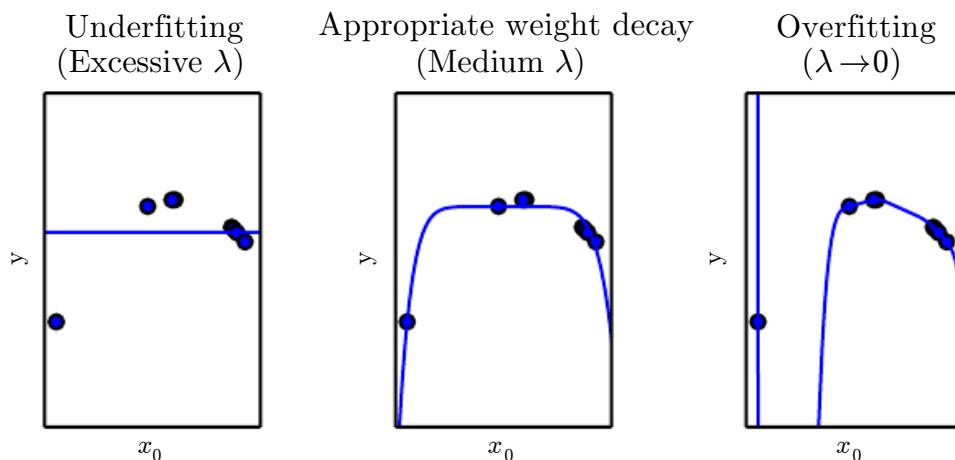


Figure 5.5: We fit a high-degree polynomial regression model to our example training set from figure 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. (Left) With very large λ , we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. (Center) With a medium value of λ , the learning algorithm recovers a curve with the right general shape. Even though the model is capable of representing functions with much more complicated shapes, weight decay has encouraged it to use a simpler function described by smaller coefficients. (Right) With weight decay approaching zero (i.e., using the Moore-Penrose pseudoinverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in figure 5.2.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because the setting is difficult to optimize. More frequently, the setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting (refer to figure 5.3). For example, we can always fit the training set better with a higher-degree polynomial and a weight decay setting of $\lambda = 0$ than we could with a lower-degree polynomial and a positive weight decay setting.

To solve this problem, we need a **validation set** of examples that the training algorithm does not observe.

Earlier we discussed how a held-out test set, composed of examples coming from the same distribution as the training set, can be used to estimate the generalization error of a learner, after the learning process has completed. It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set. Therefore, we always construct the validation set from the *training* data. Specifically, we split the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly. The subset of data used to learn the parameters is still typically called the training set, even though this may be confused with the larger pool of data used for the entire training process. The subset of data used to guide the selection of hyperparameters is called the validation set. Typically, one uses about 80 percent of the training data for training and 20 percent for validation. Since the validation set is used to “train” the hyperparameters, the validation set error will underestimate the generalization error, though typically by a smaller amount than the training error does. After all hyperparameter optimization is complete, the generalization error may be estimated using the test set.

In practice, when the same test set has been used repeatedly to evaluate performance of different algorithms over many years, and especially if we consider all the attempts from the scientific community at beating the reported state-of-the-art performance on that test set, we end up having optimistic evaluations with the test set as well. Benchmarks can thus become stale and then do not reflect the true field performance of a trained system. Thankfully, the community tends to move on to new (and usually more ambitious and larger) benchmark datasets.

5.3.1 Cross-Validation

Dividing the dataset into a fixed training set and a fixed test set can be problematic if it results in the test set being small. A small test set implies statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task.

When the dataset has hundreds of thousands of examples or more, this is not a serious issue. When the dataset is too small, alternative procedures enable one to use all the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the training and testing computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the k -fold cross-validation procedure, shown in algorithm 5.1, in which a partition of the dataset is formed by splitting it into k nonoverlapping subsets. The test error may then be estimated by taking the average test error across k trials. On trial i , the i -th subset of the data is used as the test set, and the rest of the data is used as the training set. One problem is that no unbiased estimators of the variance of such average error estimators exist (Bengio and Grandvalet, 2004), but approximations are typically used.

5.4 Estimators, Bias and Variance

The field of statistics gives us many tools to achieve the machine learning goal of solving a task not only on the training set but also to generalize. Foundational concepts such as parameter estimation, bias and variance are useful to formally characterize notions of generalization, underfitting and overfitting.

5.4.1 Point Estimation

Point estimation is the attempt to provide the single “best” prediction of some quantity of interest. In general the quantity of interest can be a single parameter or a vector of parameters in some parametric model, such as the weights in our linear regression example in section 5.1.4, but it can also be a whole function.

To distinguish estimates of parameters from their true value, our convention will be to denote a point estimate of a parameter θ by $\hat{\theta}$.

Let $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ be a set of m independent and identically distributed

Algorithm 5.1 The k -fold cross-validation algorithm. It can be used to estimate generalization error of a learning algorithm A when the given dataset \mathbb{D} is too small for a simple train/test or train/valid split to yield accurate estimation of generalization error, because the mean of a loss L on a small test set may have too high a variance. The dataset \mathbb{D} contains as elements the abstract examples $\mathbf{z}^{(i)}$ (for the i -th example), which could stand for an (input,target) pair $\mathbf{z}^{(i)} = (\mathbf{x}^{(i)}, y^{(i)})$ in the case of supervised learning, or for just an input $\mathbf{z}^{(i)} = \mathbf{x}^{(i)}$ in the case of unsupervised learning. The algorithm returns the vector of errors \mathbf{e} for each example in \mathbb{D} , whose mean is the estimated generalization error. The errors on individual examples can be used to compute a confidence interval around the mean (equation 5.47). Though these confidence intervals are not well justified after the use of cross-validation, it is still common practice to use them to declare that algorithm A is better than algorithm B only if the confidence interval of the error of algorithm A lies below and does not intersect the confidence interval of algorithm B .

Define KFoldXV(\mathbb{D}, A, L, k):

Require: \mathbb{D} , the given dataset, with elements $\mathbf{z}^{(i)}$

Require: A , the learning algorithm, seen as a function that takes a dataset as input and outputs a learned function

Require: L , the loss function, seen as a function from a learned function f and an example $\mathbf{z}^{(i)} \in \mathbb{D}$ to a scalar $\in \mathbb{R}$

Require: k , the number of folds

Split \mathbb{D} into k mutually exclusive subsets \mathbb{D}_i , whose union is \mathbb{D}

for i from 1 to k **do**

$f_i = A(\mathbb{D} \setminus \mathbb{D}_i)$

for $\mathbf{z}^{(j)}$ in \mathbb{D}_i **do**

$e_j = L(f_i, \mathbf{z}^{(j)})$

end for

end for

Return \mathbf{e}

(i.i.d.) data points. A **point estimator** or **statistic** is any function of the data:

$$\hat{\boldsymbol{\theta}}_m = g(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}). \quad (5.19)$$

The definition does not require that g return a value that is close to the true $\boldsymbol{\theta}$ or even that the range of g be the same as the set of allowable values of $\boldsymbol{\theta}$. This definition of a point estimator is very general and would enable the designer of an estimator great flexibility. While almost any function thus qualifies as an estimator,

a good estimator is a function whose output is close to the true underlying $\boldsymbol{\theta}$ that generated the training data.

For now, we take the frequentist perspective on statistics. That is, we assume that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a function of the data. Since the data is drawn from a random process, any function of the data is random. Therefore $\hat{\boldsymbol{\theta}}$ is a random variable.

Point estimation can also refer to the estimation of the relationship between input and target variables. We refer to these types of point estimates as function estimators.

Function Estimation Sometimes we are interested in performing function estimation (or function approximation). Here, we are trying to predict a variable \mathbf{y} given an input vector \mathbf{x} . We assume that there is a function $f(\mathbf{x})$ that describes the approximate relationship between \mathbf{y} and \mathbf{x} . For example, we may assume that $\mathbf{y} = f(\mathbf{x}) + \epsilon$, where ϵ stands for the part of \mathbf{y} that is not predictable from \mathbf{x} . In function estimation, we are interested in approximating f with a model or estimate \hat{f} . Function estimation is really just the same as estimating a parameter $\boldsymbol{\theta}$; the function estimator \hat{f} is simply a point estimator in function space. The linear regression example (discussed in section 5.1.4) and the polynomial regression example (discussed in section 5.2) both illustrate scenarios that may be interpreted as either estimating a parameter \mathbf{w} or estimating a function \hat{f} mapping from \mathbf{x} to y .

We now review the most commonly studied properties of point estimators and discuss what they tell us about these estimators.

5.4.2 Bias

The bias of an estimator is defined as

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}, \quad (5.20)$$

where the expectation is over the data (seen as samples from a random variable) and $\boldsymbol{\theta}$ is the true underlying value of $\boldsymbol{\theta}$ used to define the data-generating distribution. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **unbiased** if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies that $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **asymptotically unbiased** if $\lim_{m \rightarrow \infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies that $\lim_{m \rightarrow \infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Example: Bernoulli Distribution Consider a set of samples $\{x^{(1)}, \dots, x^{(m)}\}$ that are independently and identically distributed according to a Bernoulli distri-

bution with mean θ :

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})}. \quad (5.21)$$

A common estimator for the θ parameter of this distribution is the mean of the training samples:

$$\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)}. \quad (5.22)$$

To determine whether this estimator is biased, we can substitute equation 5.22 into equation 5.20:

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}[\hat{\theta}_m] - \theta \quad (5.23)$$

$$= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \theta \quad (5.24)$$

$$= \frac{1}{m} \sum_{i=1}^m \mathbb{E}[x^{(i)}] - \theta \quad (5.25)$$

$$= \frac{1}{m} \sum_{i=1}^m \sum_{x^{(i)}=0}^1 \left(x^{(i)} \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})} \right) - \theta \quad (5.26)$$

$$= \frac{1}{m} \sum_{i=1}^m (\theta) - \theta \quad (5.27)$$

$$= \theta - \theta = 0 \quad (5.28)$$

Since $\text{bias}(\hat{\theta}) = 0$, we say that our estimator $\hat{\theta}$ is unbiased.

Example: Gaussian Distribution Estimator of the Mean Now, consider a set of samples $\{x^{(1)}, \dots, x^{(m)}\}$ that are independently and identically distributed according to a Gaussian distribution $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2)$, where $i \in \{1, \dots, m\}$. Recall that the Gaussian probability density function is given by

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x^{(i)} - \mu)^2}{\sigma^2}\right). \quad (5.29)$$

A common estimator of the Gaussian mean parameter is known as the **sample mean**:

$$\hat{\mu}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)} \quad (5.30)$$

To determine the bias of the sample mean, we are again interested in calculating its expectation:

$$\text{bias}(\hat{\mu}_m) = \mathbb{E}[\hat{\mu}_m] - \mu \quad (5.31)$$

$$= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \mu \quad (5.32)$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \mathbb{E}[x^{(i)}]\right) - \mu \quad (5.33)$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \mu\right) - \mu \quad (5.34)$$

$$= \mu - \mu = 0 \quad (5.35)$$

Thus we find that the sample mean is an unbiased estimator of Gaussian mean parameter.

Example: Estimators of the Variance of a Gaussian Distribution For this example, we compare two different estimators of the variance parameter σ^2 of a Gaussian distribution. We are interested in knowing if either estimator is biased.

The first estimator of σ^2 we consider is known as the **sample variance**

$$\hat{\sigma}_m^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2, \quad (5.36)$$

where $\hat{\mu}_m$ is the sample mean. More formally, we are interested in computing

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2. \quad (5.37)$$

We begin by evaluating the term $\mathbb{E}[\hat{\sigma}_m^2]$:

$$\mathbb{E}[\hat{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2\right] \quad (5.38)$$

$$= \frac{m-1}{m} \sigma^2 \quad (5.39)$$

Returning to equation 5.37, we conclude that the bias of $\hat{\sigma}_m^2$ is $-\sigma^2/m$. Therefore, the sample variance is a biased estimator.

The **unbiased sample variance** estimator

$$\tilde{\sigma}_m^2 = \frac{1}{m-1} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2 \quad (5.40)$$

provides an alternative approach. As the name suggests this estimator is unbiased. That is, we find that $\mathbb{E}[\tilde{\sigma}_m^2] = \sigma^2$:

$$\mathbb{E}[\tilde{\sigma}_m^2] = \mathbb{E} \left[\frac{1}{m-1} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2 \right] \quad (5.41)$$

$$= \frac{m}{m-1} \mathbb{E}[\hat{\sigma}_m^2] \quad (5.42)$$

$$= \frac{m}{m-1} \left(\frac{m-1}{m} \sigma^2 \right) \quad (5.43)$$

$$= \sigma^2. \quad (5.44)$$

We have two estimators: one is biased, and the other is not. While unbiased estimators are clearly desirable, they are not always the “best” estimators. As we will see we often use biased estimators that possess other important properties.

5.4.3 Variance and Standard Error

Another property of the estimator that we might want to consider is how much we expect it to vary as a function of the data sample. Just as we computed the expectation of the estimator to determine its bias, we can compute its variance. The **variance** of an estimator is simply the variance

$$\text{Var}(\hat{\theta}) \quad (5.45)$$

where the random variable is the training set. Alternately, the square root of the variance is called the **standard error**, denoted $\text{SE}(\hat{\theta})$.

The variance, or the standard error, of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently resample the dataset from the underlying data-generating process. Just as we might like an estimator to exhibit low bias, we would also like it to have relatively low variance.

When we compute any statistic using a finite number of samples, our estimate of the true underlying parameter is uncertain, in the sense that we could have obtained other samples from the same distribution and their statistics would have

been different. The expected degree of variation in any estimator is a source of error that we want to quantify.

The standard error of the mean is given by

$$\text{SE}(\hat{\mu}_m) = \sqrt{\text{Var}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}, \quad (5.46)$$

where σ^2 is the true variance of the samples x^i . The standard error is often estimated by using an estimate of σ . Unfortunately, neither the square root of the sample variance nor the square root of the unbiased estimator of the variance provide an unbiased estimate of the standard deviation. Both approaches tend to underestimate the true standard deviation but are still used in practice. The square root of the unbiased estimator of the variance is less of an underestimate. For large m , the approximation is quite reasonable.

The standard error of the mean is very useful in machine learning experiments. We often estimate the generalization error by computing the sample mean of the error on the test set. The number of examples in the test set determines the accuracy of this estimate. Taking advantage of the central limit theorem, which tells us that the mean will be approximately distributed with a normal distribution, we can use the standard error to compute the probability that the true expectation falls in any chosen interval. For example, the 95 percent confidence interval centered on the mean $\hat{\mu}_m$ is

$$(\hat{\mu}_m - 1.96\text{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\text{SE}(\hat{\mu}_m)), \quad (5.47)$$

under the normal distribution with mean $\hat{\mu}_m$ and variance $\text{SE}(\hat{\mu}_m)^2$. In machine learning experiments, it is common to say that algorithm A is better than algorithm B if the upper bound of the 95 percent confidence interval for the error of algorithm A is less than the lower bound of the 95 percent confidence interval for the error of algorithm B .

Example: Bernoulli Distribution We once again consider a set of samples $\{x^{(1)}, \dots, x^{(m)}\}$ drawn independently and identically from a Bernoulli distribution (recall $P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1 - \theta)^{1-x^{(i)}}$). This time we are interested in computing the variance of the estimator $\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)}$.

$$\text{Var}(\hat{\theta}_m) = \text{Var}\left(\frac{1}{m} \sum_{i=1}^m x^{(i)}\right) \quad (5.48)$$

$$= \frac{1}{m^2} \sum_{i=1}^m \text{Var}(x^{(i)}) \quad (5.49)$$

$$= \frac{1}{m^2} \sum_{i=1}^m \theta(1 - \theta) \quad (5.50)$$

$$= \frac{1}{m^2} m\theta(1 - \theta) \quad (5.51)$$

$$= \frac{1}{m} \theta(1 - \theta) \quad (5.52)$$

The variance of the estimator decreases as a function of m , the number of examples in the dataset. This is a common property of popular estimators that we will return to when we discuss consistency (see section 5.4.5).

5.4.4 Trading off Bias and Variance to Minimize Mean Squared Error

Bias and variance measure two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

What happens when we are given a choice between two estimators, one with more bias and one with more variance? How do we choose between them? For example, imagine that we are interested in approximating the function shown in figure 5.2 and we are only offered the choice between a model with large bias and one that suffers from large variance. How do we choose between them?

The most common way to negotiate this trade-off is to use cross-validation. Empirically, cross-validation is highly successful on many real-world tasks. Alternatively, we can also compare the **mean squared error** (MSE) of the estimates:

$$\text{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] \quad (5.53)$$

$$= \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m) \quad (5.54)$$

The MSE measures the overall expected deviation—in a squared error sense—between the estimator and the true value of the parameter θ . As is clear from equation 5.54, evaluating the MSE incorporates both the bias and the variance. Desirable estimators are those with small MSE and these are estimators that manage to keep both their bias and variance somewhat in check.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting. When generalization

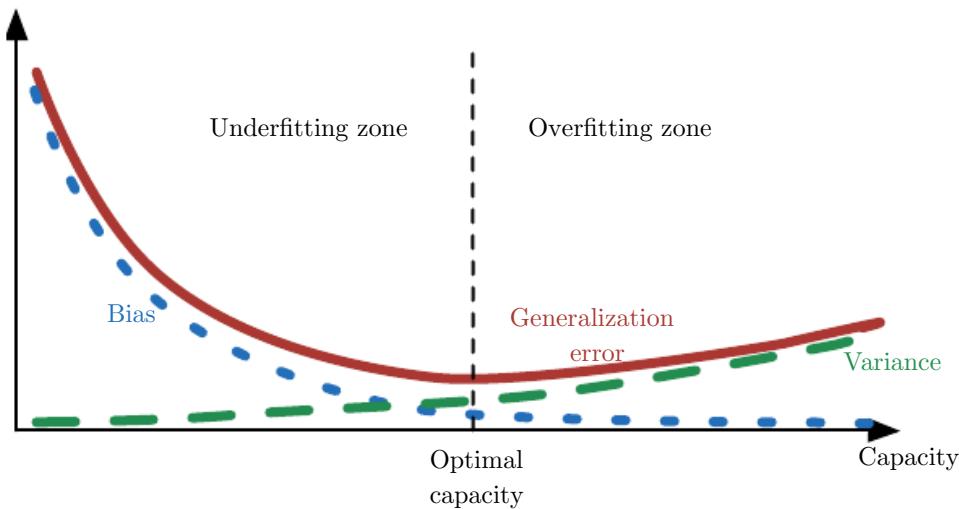


Figure 5.6: As capacity increases (x -axis), bias (dotted) tends to decrease and variance (dashed) tends to increase, yielding another U-shaped curve for generalization error (bold curve). If we vary capacity along one axis, there is an optimal capacity, with underfitting when the capacity is below this optimum and overfitting when it is above. This relationship is similar to the relationship between capacity, underfitting, and overfitting, discussed in section 5.2 and figure 5.3.

error is measured by the MSE (where bias and variance are meaningful components of generalization error), increasing capacity tends to increase variance and decrease bias. This is illustrated in figure 5.6, where we see again the U-shaped curve of generalization error as a function of capacity.

5.4.5 Consistency

So far we have discussed the properties of various estimators for a training set of fixed size. Usually, we are also concerned with the behavior of an estimator as the amount of training data grows. In particular, we usually wish that, as the number of data points m in our dataset increases, our point estimates converge to the true value of the corresponding parameters. More formally, we would like that

$$\text{plim}_{m \rightarrow \infty} \hat{\theta}_m = \theta. \quad (5.55)$$

The symbol plim indicates convergence in probability, meaning that for any $\epsilon > 0$, $P(|\hat{\theta}_m - \theta| > \epsilon) \rightarrow 0$ as $m \rightarrow \infty$. The condition described by equation 5.55 is known as **consistency**. It is sometimes referred to as weak consistency, with strong consistency referring to the **almost sure** convergence of $\hat{\theta}$ to θ . **Almost**

sure convergence of a sequence of random variables $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots$ to a value \mathbf{x} occurs when $p(\lim_{m \rightarrow \infty} \mathbf{x}^{(m)} = \mathbf{x}) = 1$.

Consistency ensures that the bias induced by the estimator diminishes as the number of data examples grows. However, the reverse is not true—asymptotic unbiasedness does not imply consistency. For example, consider estimating the mean parameter μ of a normal distribution $\mathcal{N}(x; \mu, \sigma^2)$, with a dataset consisting of m samples: $\{x^{(1)}, \dots, x^{(m)}\}$. We could use the first sample $x^{(1)}$ of the dataset as an unbiased estimator: $\hat{\theta} = x^{(1)}$. In that case, $\mathbb{E}(\hat{\theta}_m) = \theta$, so the estimator is unbiased no matter how many data points are seen. This, of course, implies that the estimate is asymptotically unbiased. However, this is not a consistent estimator as it is *not* the case that $\hat{\theta}_m \rightarrow \theta$ as $m \rightarrow \infty$.

5.5 Maximum Likelihood Estimation

We have seen some definitions of common estimators and analyzed their properties. But where did these estimators come from? Rather than guessing that some function might make a good estimator and then analyzing its bias and variance, we would like to have some principle from which we can derive specific functions that are good estimators for different models.

The most common such principle is the maximum likelihood principle.

Consider a set of m examples $\mathbb{X} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ drawn independently from the true but unknown data-generating distribution $p_{\text{data}}(\mathbf{x})$.

Let $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ be a parametric family of probability distributions over the same space indexed by $\boldsymbol{\theta}$. In other words, $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ maps any configuration \mathbf{x} to a real number estimating the true probability $p_{\text{data}}(\mathbf{x})$.

The maximum likelihood estimator for $\boldsymbol{\theta}$ is then defined as

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}), \quad (5.56)$$

$$= \arg \max_{\boldsymbol{\theta}} \prod_{i=1}^m p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.57)$$

This product over many probabilities can be inconvenient for various reasons. For example, it is prone to numerical underflow. To obtain a more convenient but equivalent optimization problem, we observe that taking the logarithm of the likelihood does not change its arg max but does conveniently transform a product

into a sum:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.58)$$

Because the arg max does not change when we rescale the cost function, we can divide by m to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution \hat{p}_{data} defined by the training data:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}). \quad (5.59)$$

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution \hat{p}_{data} , defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. The KL divergence is given by

$$D_{\text{KL}}(\hat{p}_{\text{data}} \| p_{\text{model}}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log \hat{p}_{\text{data}}(\mathbf{x}) - \log p_{\text{model}}(\mathbf{x})]. \quad (5.60)$$

The term on the left is a function only of the data-generating process, not the model. This means when we train the model to minimize the KL divergence, we need only minimize

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log p_{\text{model}}(\mathbf{x})], \quad (5.61)$$

which is of course the same as the maximization in equation 5.59.

Minimizing this KL divergence corresponds exactly to minimizing the cross-entropy between the distributions. Many authors use the term “cross-entropy” to identify specifically the negative log-likelihood of a Bernoulli or softmax distribution, but that is a misnomer. Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by model. For example, mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.

We can thus see maximum likelihood as an attempt to make the model distribution match the empirical distribution \hat{p}_{data} . Ideally, we would like to match the true data-generating distribution p_{data} , but we have no direct access to this distribution.

While the optimal $\boldsymbol{\theta}$ is the same regardless of whether we are maximizing the likelihood or minimizing the KL divergence, the values of the objective functions are different. In software, we often phrase both as minimizing a cost function. Maximum likelihood thus becomes minimization of the negative log-likelihood (NLL), or equivalently, minimization of the cross-entropy. The perspective of maximum likelihood as minimum KL divergence becomes helpful in this case because the KL divergence has a known minimum value of zero. The negative log-likelihood can actually become negative when \mathbf{x} is real-valued.

5.5.1 Conditional Log-Likelihood and Mean Squared Error

The maximum likelihood estimator can readily be generalized to estimate a conditional probability $P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ in order to predict \mathbf{y} given \mathbf{x} . This is actually the most common situation because it forms the basis for most supervised learning. If \mathbf{X} represents all our inputs and \mathbf{Y} all our observed targets, then the conditional maximum likelihood estimator is

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} P(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}). \quad (5.62)$$

If the examples are assumed to be i.i.d., then this can be decomposed into

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.63)$$

Example: Linear Regression as Maximum Likelihood Linear regression, introduced in section 5.1.4, may be justified as a maximum likelihood procedure. Previously, we motivated linear regression as an algorithm that learns to take an input \mathbf{x} and produce an output value \hat{y} . The mapping from \mathbf{x} to \hat{y} is chosen to minimize mean squared error, a criterion that we introduced more or less arbitrarily. We now revisit linear regression from the point of view of maximum likelihood estimation. Instead of producing a single prediction \hat{y} , we now think of the model as producing a conditional distribution $p(y \mid \mathbf{x})$. We can imagine that with an infinitely large training set, we might see several training examples with the same input value \mathbf{x} but different values of y . The goal of the learning algorithm is now to fit the distribution $p(y \mid \mathbf{x})$ to all those different y values that are all compatible with \mathbf{x} . To derive the same linear regression algorithm we obtained before, we define $p(y \mid \mathbf{x}) = \mathcal{N}(y; \hat{y}(\mathbf{x}; \mathbf{w}), \sigma^2)$. The function $\hat{y}(\mathbf{x}; \mathbf{w})$ gives the prediction of the mean of the Gaussian. In this example, we assume that the variance is fixed to some constant σ^2 chosen by the user. We will see that this choice of the functional form of $p(y \mid \mathbf{x})$ causes the maximum likelihood estimation procedure to yield the same learning algorithm as we developed before. Since the examples are assumed to be i.i.d., the conditional log-likelihood (equation 5.63) is given by

$$\sum_{i=1}^m \log p(y^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta}) \quad (5.64)$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^m \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}, \quad (5.65)$$

where $\hat{y}^{(i)}$ is the output of the linear regression on the i -th input $\mathbf{x}^{(i)}$ and m is the number of the training examples. Comparing the log-likelihood with the mean squared error,

$$\text{MSE}_{\text{train}} = \frac{1}{m} \sum_{i=1}^m \|\hat{y}^{(i)} - y^{(i)}\|^2, \quad (5.66)$$

we immediately see that maximizing the log-likelihood with respect to \mathbf{w} yields the same estimate of the parameters \mathbf{w} as does minimizing the mean squared error. The two criteria have different values but the same location of the optimum. This justifies the use of the MSE as a maximum likelihood estimation procedure. As we will see, the maximum likelihood estimator has several desirable properties.

5.5.2 Properties of Maximum Likelihood

The main appeal of the maximum likelihood estimator is that it can be shown to be the best estimator asymptotically, as the number of examples $m \rightarrow \infty$, in terms of its rate of convergence as m increases.

Under appropriate conditions, the maximum likelihood estimator has the property of consistency (see section 5.4.5), meaning that as the number of training examples approaches infinity, the maximum likelihood estimate of a parameter converges to the true value of the parameter. These conditions are as follows:

- The true distribution p_{data} must lie within the model family $p_{\text{model}}(\cdot; \boldsymbol{\theta})$. Otherwise, no estimator can recover p_{data} .
- The true distribution p_{data} must correspond to exactly one value of $\boldsymbol{\theta}$. Otherwise, maximum likelihood can recover the correct p_{data} but will not be able to determine which value of $\boldsymbol{\theta}$ was used by the data-generating process.

There are other inductive principles besides the maximum likelihood estimator, many of which share the property of being consistent estimators. Consistent estimators can differ, however, in their **statistical efficiency**, meaning that one consistent estimator may obtain lower generalization error for a fixed number of samples m , or equivalently, may require fewer examples to obtain a fixed level of generalization error.

Statistical efficiency is typically studied in the **parametric case** (as in linear regression), where our goal is to estimate the value of a parameter (assuming it is possible to identify the true parameter), not the value of a function. A way to measure how close we are to the true parameter is by the expected mean squared error, computing the squared difference between the estimated and true parameter

values, where the expectation is over m training samples from the data-generating distribution. That parametric mean squared error decreases as m increases, and for m large, the Cramér-Rao lower bound (Rao, 1945; Cramér, 1946) shows that no consistent estimator has a lower MSE than the maximum likelihood estimator.

For these reasons (consistency and efficiency), maximum likelihood is often considered the preferred estimator to use for machine learning. When the number of examples is small enough to yield overfitting behavior, regularization strategies such as weight decay may be used to obtain a biased version of maximum likelihood that has less variance when training data is limited.

5.6 Bayesian Statistics

So far we have discussed **frequentist statistics** and approaches based on estimating a single value of $\boldsymbol{\theta}$, then making all predictions thereafter based on that one estimate. Another approach is to consider all possible values of $\boldsymbol{\theta}$ when making a prediction. The latter is the domain of **Bayesian statistics**.

As discussed in section 5.4.1, the frequentist perspective is that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a random variable on account of it being a function of the dataset (which is seen as random).

The Bayesian perspective on statistics is quite different. The Bayesian uses probability to reflect degrees of certainty in states of knowledge. The dataset is directly observed and so is not random. On the other hand, the true parameter $\boldsymbol{\theta}$ is unknown or uncertain and thus is represented as a random variable.

Before observing the data, we represent our knowledge of $\boldsymbol{\theta}$ using the **prior probability distribution**, $p(\boldsymbol{\theta})$ (sometimes referred to as simply “the prior”). Generally, the machine learning practitioner selects a prior distribution that is quite broad (i.e., with high entropy) to reflect a high degree of uncertainty in the value of $\boldsymbol{\theta}$ before observing any data. For example, one might assume a prior that $\boldsymbol{\theta}$ lies in some finite range or volume, with a uniform distribution. Many priors instead reflect a preference for “simpler” solutions (such as smaller magnitude coefficients, or a function that is closer to being constant).

Now consider that we have a set of data samples $\{x^{(1)}, \dots, x^{(m)}\}$. We can recover the effect of data on our belief about $\boldsymbol{\theta}$ by combining the data likelihood $p(x^{(1)}, \dots, x^{(m)} | \boldsymbol{\theta})$ with the prior via Bayes’ rule:

$$p(\boldsymbol{\theta} | x^{(1)}, \dots, x^{(m)}) = \frac{p(x^{(1)}, \dots, x^{(m)} | \boldsymbol{\theta})p(\boldsymbol{\theta})}{p(x^{(1)}, \dots, x^{(m)})} \quad (5.67)$$

In the scenarios where Bayesian estimation is typically used, the prior begins as a relatively uniform or Gaussian distribution with high entropy, and the observation of the data usually causes the posterior to lose entropy and concentrate around a few highly likely values of the parameters.

Relative to maximum likelihood estimation, Bayesian estimation offers two important differences. First, unlike the maximum likelihood approach that makes predictions using a point estimate of $\boldsymbol{\theta}$, the Bayesian approach is to make predictions using a full distribution over $\boldsymbol{\theta}$. For example, after observing m examples, the predicted distribution over the next data sample, $x^{(m+1)}$, is given by

$$p(x^{(m+1)} | x^{(1)}, \dots, x^{(m)}) = \int p(x^{(m+1)} | \boldsymbol{\theta}) p(\boldsymbol{\theta} | x^{(1)}, \dots, x^{(m)}) d\boldsymbol{\theta}. \quad (5.68)$$

Here each value of $\boldsymbol{\theta}$ with positive probability density contributes to the prediction of the next example, with the contribution weighted by the posterior density itself. After having observed $\{x^{(1)}, \dots, x^{(m)}\}$, if we are still quite uncertain about the value of $\boldsymbol{\theta}$, then this uncertainty is incorporated directly into any predictions we might make.

In section 5.4, we discussed how the frequentist approach addresses the uncertainty in a given point estimate of $\boldsymbol{\theta}$ by evaluating its variance. The variance of the estimator is an assessment of how the estimate might change with alternative samplings of the observed data. The Bayesian answer to the question of how to deal with the uncertainty in the estimator is to simply integrate over it, which tends to protect well against overfitting. This integral is of course just an application of the laws of probability, making the Bayesian approach simple to justify, while the frequentist machinery for constructing an estimator is based on the rather ad hoc decision to summarize all knowledge contained in the dataset with a single point estimate.

The second important difference between the Bayesian approach to estimation and the maximum likelihood approach is due to the contribution of the Bayesian prior distribution. The prior has an influence by shifting probability mass density towards regions of the parameter space that are preferred a priori. In practice, the prior often expresses a preference for models that are simpler or more smooth. Critics of the Bayesian approach identify the prior as a source of subjective human judgment affecting the predictions.

Bayesian methods typically generalize much better when limited training data is available but typically suffer from high computational cost when the number of training examples is large.

Example: Bayesian Linear Regression Here we consider the Bayesian estimation approach to learning the linear regression parameters. In linear regression, we learn a linear mapping from an input vector $\mathbf{x} \in \mathbb{R}^n$ to predict the value of a scalar $y \in \mathbb{R}$. The prediction is parametrized by the vector $\mathbf{w} \in \mathbb{R}^n$:

$$\hat{y} = \mathbf{w}^\top \mathbf{x}. \quad (5.69)$$

Given a set of m training samples $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$, we can express the prediction of y over the entire training set as

$$\hat{\mathbf{y}}^{(\text{train})} = \mathbf{X}^{(\text{train})} \mathbf{w}. \quad (5.70)$$

Expressed as a Gaussian conditional distribution on $\mathbf{y}^{(\text{train})}$, we have

$$p(\mathbf{y}^{(\text{train})} | \mathbf{X}^{(\text{train})}, \mathbf{w}) = \mathcal{N}(\mathbf{y}^{(\text{train})}; \mathbf{X}^{(\text{train})} \mathbf{w}, \mathbf{I}) \quad (5.71)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y}^{(\text{train})} - \mathbf{X}^{(\text{train})} \mathbf{w})^\top (\mathbf{y}^{(\text{train})} - \mathbf{X}^{(\text{train})} \mathbf{w})\right), \quad (5.72)$$

where we follow the standard MSE formulation in assuming that the Gaussian variance on y is one. In what follows, to reduce the notational burden, we refer to $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ as simply (\mathbf{X}, \mathbf{y}) .

To determine the posterior distribution over the model parameter vector \mathbf{w} , we first need to specify a prior distribution. The prior should reflect our naive belief about the value of these parameters. While it is sometimes difficult or unnatural to express our prior beliefs in terms of the parameters of the model, in practice we typically assume a fairly broad distribution, expressing a high degree of uncertainty about θ . For real-valued parameters it is common to use a Gaussian as a prior distribution,

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right), \quad (5.73)$$

where $\boldsymbol{\mu}_0$ and $\boldsymbol{\Lambda}_0$ are the prior distribution mean vector and covariance matrix respectively.¹

With the prior thus specified, we can now proceed in determining the **posterior distribution** over the model parameters:

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} | \mathbf{X}, \mathbf{w}) p(\mathbf{w}) \quad (5.74)$$

¹Unless there is a reason to use a particular covariance structure, we typically assume a diagonal covariance matrix $\boldsymbol{\Lambda}_0 = \text{diag}(\boldsymbol{\lambda}_0)$.

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})\right) \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right) \quad (5.75)$$

$$\propto \exp\left(-\frac{1}{2}\left(-2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \boldsymbol{\Lambda}_0^{-1}\mathbf{w} - 2\boldsymbol{\mu}_0^\top \boldsymbol{\Lambda}_0^{-1}\mathbf{w}\right)\right). \quad (5.76)$$

We now define $\boldsymbol{\Lambda}_m = (\mathbf{X}^\top \mathbf{X} + \boldsymbol{\Lambda}_0^{-1})^{-1}$ and $\boldsymbol{\mu}_m = \boldsymbol{\Lambda}_m (\mathbf{X}^\top \mathbf{y} + \boldsymbol{\Lambda}_0^{-1} \boldsymbol{\mu}_0)$. Using these new variables, we find that the posterior may be rewritten as a Gaussian distribution:

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\mathbf{w} - \boldsymbol{\mu}_m) + \frac{1}{2}\boldsymbol{\mu}_m^\top \boldsymbol{\Lambda}_m^{-1}\boldsymbol{\mu}_m\right) \quad (5.77)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\mathbf{w} - \boldsymbol{\mu}_m)\right). \quad (5.78)$$

All terms that do not include the parameter vector \mathbf{w} have been omitted; they are implied by the fact that the distribution must be normalized to integrate to 1. Equation 3.23 shows how to normalize a multivariate Gaussian distribution.

Examining this posterior distribution enables us to gain some intuition for the effect of Bayesian inference. In most situations, we set $\boldsymbol{\mu}_0$ to $\mathbf{0}$. If we set $\boldsymbol{\Lambda}_0 = \frac{1}{\alpha} \mathbf{I}$, then $\boldsymbol{\mu}_m$ gives the same estimate of \mathbf{w} as does frequentist linear regression with a weight decay penalty of $\alpha \mathbf{w}^\top \mathbf{w}$. One difference is that the Bayesian estimate is undefined if α is set to zero—we are not allowed to begin the Bayesian learning process with an infinitely wide prior on \mathbf{w} . The more important difference is that the Bayesian estimate provides a covariance matrix, showing how likely all the different values of \mathbf{w} are, rather than providing only the estimate $\boldsymbol{\mu}_m$.

5.6.1 Maximum a Posteriori (MAP) Estimation

While the most principled approach is to make predictions using the full Bayesian posterior distribution over the parameter $\boldsymbol{\theta}$, it is still often desirable to have a single point estimate. One common reason for desiring a point estimate is that most operations involving the Bayesian posterior for most interesting models are intractable, and a point estimate offers a tractable approximation. Rather than simply returning to the maximum likelihood estimate, we can still gain some of the benefit of the Bayesian approach by allowing the prior to influence the choice of the point estimate. One rational way to do this is to choose the **maximum a posteriori** (MAP) point estimate. The MAP estimate chooses the point of

maximal posterior probability (or maximal probability density in the more common case of continuous $\boldsymbol{\theta}$):

$$\boldsymbol{\theta}_{\text{MAP}} = \arg \max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathbf{x}) = \arg \max_{\boldsymbol{\theta}} \log p(\mathbf{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}). \quad (5.79)$$

We recognize, on the righthand side, $\log p(\mathbf{x} \mid \boldsymbol{\theta})$, that is, the standard log-likelihood term, and $\log p(\boldsymbol{\theta})$, corresponding to the prior distribution.

As an example, consider a linear regression model with a Gaussian prior on the weights \mathbf{w} . If this prior is given by $\mathcal{N}(\mathbf{w}; \mathbf{0}, \frac{1}{\lambda} \mathbf{I}^2)$, then the log-prior term in equation 5.79 is proportional to the familiar $\lambda \mathbf{w}^\top \mathbf{w}$ weight decay penalty, plus a term that does not depend on \mathbf{w} and does not affect the learning process. MAP Bayesian inference with a Gaussian prior on the weights thus corresponds to weight decay.

As with full Bayesian inference, MAP Bayesian inference has the advantage of leveraging information that is brought by the prior and cannot be found in the training data. This additional information helps to reduce the variance in the MAP point estimate (in comparison to the ML estimate). However, it does so at the price of increased bias.

Many regularized estimation strategies, such as maximum likelihood learning regularized with weight decay, can be interpreted as making the MAP approximation to Bayesian inference. This view applies when the regularization consists of adding an extra term to the objective function that corresponds to $\log p(\boldsymbol{\theta})$. Not all regularization penalties correspond to MAP Bayesian inference. For example, some regularizer terms may not be the logarithm of a probability distribution. Other regularization terms depend on the data, which of course a prior probability distribution is not allowed to do.

MAP Bayesian inference provides a straightforward way to design complicated yet interpretable regularization terms. For example, a more complicated penalty term can be derived by using a mixture of Gaussians, rather than a single Gaussian distribution, as the prior (Nowlan and Hinton, 1992).

5.7 Supervised Learning Algorithms

Recall from section 5.1.3 that supervised learning algorithms are, roughly speaking, learning algorithms that learn to associate some input with some output, given a training set of examples of inputs \mathbf{x} and outputs \mathbf{y} . In many cases the outputs \mathbf{y} may be difficult to collect automatically and must be provided by a human

“supervisor,” but the term still applies even when the training set targets were collected automatically.

5.7.1 Probabilistic Supervised Learning

Most supervised learning algorithms in this book are based on estimating a probability distribution $p(y | \mathbf{x})$. We can do this simply by using maximum likelihood estimation to find the best parameter vector $\boldsymbol{\theta}$ for a parametric family of distributions $p(y | \mathbf{x}; \boldsymbol{\theta})$.

We have already seen that linear regression corresponds to the family

$$p(y | \mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}(y; \boldsymbol{\theta}^\top \mathbf{x}, \mathbf{I}). \quad (5.80)$$

We can generalize linear regression to the classification scenario by defining a different family of probability distributions. If we have two classes, class 0 and class 1, then we need only specify the probability of one of these classes. The probability of class 1 determines the probability of class 0, because these two values must add up to 1.

The normal distribution over real-valued numbers that we used for linear regression is parametrized in terms of a mean. Any value we supply for this mean is valid. A distribution over a binary variable is slightly more complicated, because its mean must always be between 0 and 1. One way to solve this problem is to use the logistic sigmoid function to squash the output of the linear function into the interval (0, 1) and interpret that value as a probability:

$$p(y = 1 | \mathbf{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x}). \quad (5.81)$$

This approach is known as **logistic regression** (a somewhat strange name since we use the model for classification rather than regression).

In the case of linear regression, we were able to find the optimal weights by solving the normal equations. Logistic regression is somewhat more difficult. There is no closed-form solution for its optimal weights. Instead, we must search for them by maximizing the log-likelihood. We can do this by minimizing the negative log-likelihood using gradient descent.

This same strategy can be applied to essentially any supervised learning problem, by writing down a parametric family of conditional probability distributions over the right kind of input and output variables.

5.7.2 Support Vector Machines

One of the most influential approaches to supervised learning is the support vector machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995). This model is similar to logistic regression in that it is driven by a linear function $\mathbf{w}^\top \mathbf{x} + b$. Unlike logistic regression, the support vector machine does not provide probabilities, but only outputs a class identity. The SVM predicts that the positive class is present when $\mathbf{w}^\top \mathbf{x} + b$ is positive. Likewise, it predicts that the negative class is present when $\mathbf{w}^\top \mathbf{x} + b$ is negative.

One key innovation associated with support vector machines is the **kernel trick**. The kernel trick consists of observing that many machine learning algorithms can be written exclusively in terms of dot products between examples. For example, it can be shown that the linear function used by the support vector machine can be re-written as

$$\mathbf{w}^\top \mathbf{x} + b = b + \sum_{i=1}^m \alpha_i \mathbf{x}^\top \mathbf{x}^{(i)}, \quad (5.82)$$

where $\mathbf{x}^{(i)}$ is a training example, and $\boldsymbol{\alpha}$ is a vector of coefficients. Rewriting the learning algorithm this way enables us to replace \mathbf{x} with the output of a given feature function $\phi(\mathbf{x})$ and the dot product with a function $k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}^{(i)})$ called a **kernel**. The \cdot operator represents an inner product analogous to $\phi(\mathbf{x})^\top \phi(\mathbf{x}^{(i)})$. For some feature spaces, we may not use literally the vector inner product. In some infinite dimensional spaces, we need to use other kinds of inner products, for example, inner products based on integration rather than summation. A complete development of these kinds of inner products is beyond the scope of this book.

After replacing dot products with kernel evaluations, we can make predictions using the function

$$f(\mathbf{x}) = b + \sum_i \alpha_i k(\mathbf{x}, \mathbf{x}^{(i)}). \quad (5.83)$$

This function is nonlinear with respect to \mathbf{x} , but the relationship between $\phi(\mathbf{x})$ and $f(\mathbf{x})$ is linear. Also, the relationship between $\boldsymbol{\alpha}$ and $f(\mathbf{x})$ is linear. The kernel-based function is exactly equivalent to preprocessing the data by applying $\phi(\mathbf{x})$ to all inputs, then learning a linear model in the new transformed space.

The kernel trick is powerful for two reasons. First, it enables us to learn models that are nonlinear as a function of \mathbf{x} using convex optimization techniques that are guaranteed to converge efficiently. This is possible because we consider ϕ fixed and optimize only $\boldsymbol{\alpha}$, that is, the optimization algorithm can view the decision function as being linear in a different space. Second, the kernel function k often

admits an implementation that is significantly more computationally efficient than naively constructing two $\phi(\mathbf{x})$ vectors and explicitly taking their dot product.

In some cases, $\phi(\mathbf{x})$ can even be infinite dimensional, which would result in an infinite computational cost for the naive, explicit approach. In many cases, $k(\mathbf{x}, \mathbf{x}')$ is a nonlinear, tractable function of \mathbf{x} even when $\phi(\mathbf{x})$ is intractable. As an example of an infinite-dimensional feature space with a tractable kernel, we construct a feature mapping $\phi(x)$ over the nonnegative integers x . Suppose that this mapping returns a vector containing x ones followed by infinitely many zeros. We can write a kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$ that is exactly equivalent to the corresponding infinite-dimensional dot product.

The most commonly used kernel is the **Gaussian kernel**,

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}), \quad (5.84)$$

where $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the standard normal density. This kernel is also known as the **radial basis function** (RBF) kernel, because its value decreases along lines in \mathbf{v} space radiating outward from \mathbf{u} . The Gaussian kernel corresponds to a dot product in an infinite-dimensional space, but the derivation of this space is less straightforward than in our example of the min kernel over the integers.

We can think of the Gaussian kernel as performing a kind of **template matching**. A training example \mathbf{x} associated with training label y becomes a template for class y . When a test point \mathbf{x}' is near \mathbf{x} according to Euclidean distance, the Gaussian kernel has a large response, indicating that \mathbf{x}' is very similar to the \mathbf{x} template. The model then puts a large weight on the associated training label y . Overall, the prediction will combine many such training labels weighted by the similarity of the corresponding training examples.

Support vector machines are not the only algorithm that can be enhanced using the kernel trick. Many other linear models can be enhanced in this way. The category of algorithms that employ the kernel trick is known as **kernel machines**, or **kernel methods** (Williams and Rasmussen, 1996; Schölkopf *et al.*, 1999).

A major drawback to kernel machines is that the cost of evaluating the decision function is linear in the number of training examples, because the i -th example contributes a term $\alpha_i k(\mathbf{x}, \mathbf{x}^{(i)})$ to the decision function. Support vector machines are able to mitigate this by learning an $\boldsymbol{\alpha}$ vector that contains mostly zeros. Classifying a new example then requires evaluating the kernel function only for the training examples that have nonzero α_i . These training examples are known as **support vectors**.

Kernel machines also suffer from a high computational cost of training when the dataset is large. We revisit this idea in section 5.9. Kernel machines with

generic kernels struggle to generalize well. We explain why in section 5.11. The modern incarnation of deep learning was designed to overcome these limitations of kernel machines. The current deep learning renaissance began when Hinton *et al.* (2006) demonstrated that a neural network could outperform the RBF kernel SVM on the MNIST benchmark.

5.7.3 Other Simple Supervised Learning Algorithms

We have already briefly encountered another nonprobabilistic supervised learning algorithm, nearest neighbor regression. More generally, k -nearest neighbors is a family of techniques that can be used for classification or regression. As a nonparametric learning algorithm, k -nearest neighbors is not restricted to a fixed number of parameters. We usually think of the k -nearest neighbors algorithm as not having any parameters but rather implementing a simple function of the training data. In fact, there is not even really a training stage or learning process. Instead, at test time, when we want to produce an output y for a new test input \mathbf{x} , we find the k -nearest neighbors to \mathbf{x} in the training data \mathbf{X} . We then return the average of the corresponding y values in the training set. This works for essentially any kind of supervised learning where we can define an average over y values. In the case of classification, we can average over one-hot code vectors \mathbf{c} with $c_y = 1$ and $c_i = 0$ for all other values of i . We can then interpret the average over these one-hot codes as giving a probability distribution over classes. As a nonparametric learning algorithm, k -nearest neighbor can achieve very high capacity. For example, suppose we have a multiclass classification task and measure performance with 0-1 loss. In this setting, 1-nearest neighbor converges to double the Bayes error as the number of training examples approaches infinity. The error in excess of the Bayes error results from choosing a single neighbor by breaking ties between equally distant neighbors randomly. When there is infinite training data, all test points \mathbf{x} will have infinitely many training set neighbors at distance zero. If we allow the algorithm to use all these neighbors to vote, rather than randomly choosing one of them, the procedure converges to the Bayes error rate. The high capacity of k -nearest neighbors enables it to obtain high accuracy given a large training set. It does so at high computational cost, however, and it may generalize very badly given a small finite training set. One weakness of k -nearest neighbors is that it cannot learn that one feature is more discriminative than another. For example, imagine we have a regression task with $\mathbf{x} \in \mathbb{R}^{100}$ drawn from an isotropic Gaussian distribution, but only a single variable x_1 is relevant to the output. Suppose further that this feature simply encodes the output directly, that $y = x_1$ in all cases. Nearest neighbor regression will not be able to detect this simple pattern.

The nearest neighbor of most points \mathbf{x} will be determined by the large number of features x_2 through x_{100} , not by the lone feature x_1 . Thus the output on small training sets will essentially be random.

Another type of learning algorithm that also breaks the input space into regions and has separate parameters for each region is the **decision tree** (Breiman *et al.*, 1984) and its many variants. As shown in figure 5.7, each node of the decision tree is associated with a region in the input space, and internal nodes break that region into one subregion for each child of the node (typically using an axis-aligned cut). Space is thus subdivided into nonoverlapping regions, with a one-to-one correspondence between leaf nodes and input regions. Each leaf node usually maps every point in its input region to the same output. Decision trees are usually trained with specialized algorithms that are beyond the scope of this book. The learning algorithm can be considered nonparametric if it is allowed to learn a tree of arbitrary size, though decision trees are usually regularized with size constraints that turn them into parametric models in practice. Decision trees as they are typically used, with axis-aligned splits and constant outputs within each node, struggle to solve some problems that are easy even for logistic regression. For example, if we have a two-class problem, and the positive class occurs wherever $x_2 > x_1$, the decision boundary is not axis aligned. The decision tree will thus need to approximate the decision boundary with many nodes, implementing a step function that constantly walks back and forth across the true decision function with axis-aligned steps.

As we have seen, nearest neighbor predictors and decision trees have many limitations. Nonetheless, they are useful learning algorithms when computational resources are constrained. We can also build intuition for more sophisticated learning algorithms by thinking about the similarities and differences between sophisticated algorithms and k -nearest neighbors or decision tree baselines.

See Murphy (2012), Bishop (2006), Hastie *et al.* (2001) or other machine learning textbooks for more material on traditional supervised learning algorithms.

5.8 Unsupervised Learning Algorithms

Recall from section 5.1.3 that unsupervised algorithms are those that experience only “features” but not a supervision signal. The distinction between supervised and unsupervised algorithms is not formally and rigidly defined because there is no objective test for distinguishing whether a value is a feature or a target provided by a supervisor. Informally, unsupervised learning refers to most attempts to extract information from a distribution that do not require human labor to annotate

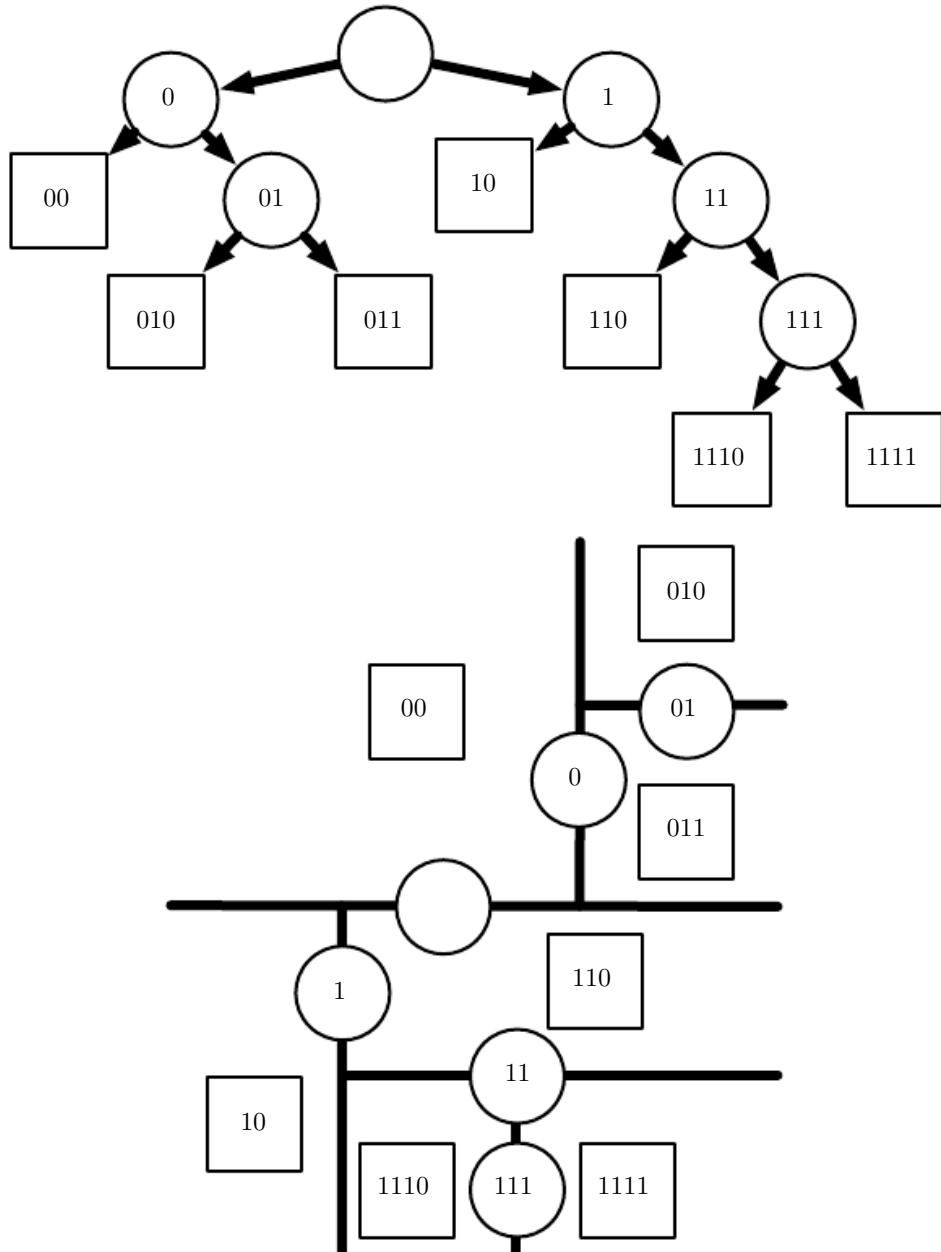


Figure 5.7: Diagrams describing how a decision tree works. (*Top*) Each node of the tree chooses to send the input example to the child node on the left (0) or to the child node on the right (1). Internal nodes are drawn as circles and leaf nodes as squares. Each node is displayed with a binary string identifier corresponding to its position in the tree, obtained by appending a bit to its parent identifier (0 = choose left or top, 1 = choose right or bottom). (*Bottom*) The tree divides space into regions. The 2-D plane shows how a decision tree might divide \mathbb{R}^2 . The nodes of the tree are plotted in this plane, with each internal node drawn along the dividing line it uses to categorize examples, and leaf nodes drawn in the center of the region of examples they receive. The result is a piecewise-constant function, with one piece per leaf. Each leaf requires at least one training example to define, so it is not possible for the decision tree to learn a function that has more local maxima than the number of training examples.

examples. The term is usually associated with density estimation, learning to draw samples from a distribution, learning to denoise data from some distribution, finding a manifold that the data lies near, or clustering the data into groups of related examples.

A classic unsupervised learning task is to find the “best” representation of the data. By “best” we can mean different things, but generally speaking we are looking for a representation that preserves as much information about \mathbf{x} as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than \mathbf{x} itself.

There are multiple ways of defining a simpler representation. Three of the most common include lower-dimensional representations, sparse representations, and independent representations. Low-dimensional representations attempt to compress as much information about x as possible in a smaller representation. Sparse representations (Barlow, 1989; Olshausen and Field, 1996; Hinton and Ghahramani, 1997) embed the dataset into a representation whose entries are mostly zeros for most inputs. The use of sparse representations typically requires increasing the dimensionality of the representation, so that the representation becoming mostly zeros does not discard too much information. This results in an overall structure of the representation that tends to distribute data along the axes of the representation space. Independent representations attempt to *disentangle* the sources of variation underlying the data distribution such that the dimensions of the representation are statistically independent.

Of course these three criteria are certainly not mutually exclusive. Low-dimensional representations often yield elements that have fewer or weaker dependencies than the original high-dimensional data. This is because one way to reduce the size of a representation is to find and remove redundancies. Identifying and removing more redundancy enables the dimensionality reduction algorithm to achieve more compression while discarding less information.

The notion of representation is one of the central themes of deep learning and therefore one of the central themes in this book. In this section, we develop some simple examples of representation learning algorithms. Together, these example algorithms show how to operationalize all three of the criteria above. Most of the remaining chapters introduce additional representation learning algorithms that develop these criteria in different ways or introduce other criteria.

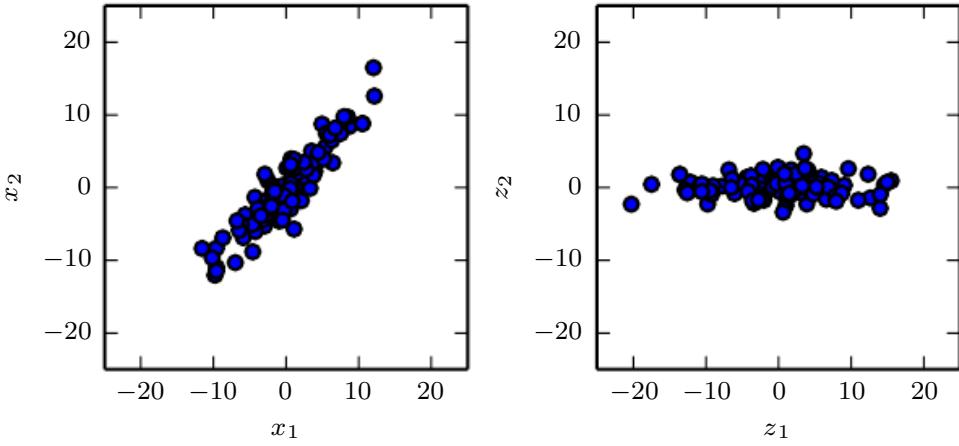


Figure 5.8: PCA learns a linear projection that aligns the direction of greatest variance with the axes of the new space. (*Left*) The original data consist of samples of \mathbf{x} . In this space, the variance might occur along directions that are not axis aligned. (*Right*) The transformed data $\mathbf{z} = \mathbf{x}^\top \mathbf{W}$ now varies most along the axis z_1 . The direction of second-most variance is now along z_2 .

5.8.1 Principal Components Analysis

In section 2.12, we saw that the principal components analysis algorithm provides a means of compressing data. We can also view PCA as an unsupervised learning algorithm that learns a representation of data. This representation is based on two of the criteria for a simple representation described above. PCA learns a representation that has lower dimensionality than the original input. It also learns a representation whose elements have no linear correlation with each other. This is a first step toward the criterion of learning representations whose elements are statistically independent. To achieve full independence, a representation learning algorithm must also remove the nonlinear relationships between variables.

PCA learns an orthogonal, linear transformation of the data that projects an input \mathbf{x} to a representation \mathbf{z} as shown in figure 5.8. In section 2.12, we saw that we could learn a one-dimensional representation that best reconstructs the original data (in the sense of mean squared error) and that this representation actually corresponds to the first principal component of the data. Thus we can use PCA as a simple and effective dimensionality reduction method that preserves as much of the information in the data as possible (again, as measured by least-squares reconstruction error). In the following, we will study how the PCA representation decorrelates the original data representation \mathbf{X} .

Let us consider the $m \times n$ design matrix \mathbf{X} . We will assume that the data has

a mean of zero, $\mathbb{E}[\mathbf{x}] = \mathbf{0}$. If this is not the case, the data can easily be centered by subtracting the mean from all examples in a preprocessing step.

The unbiased sample covariance matrix associated with \mathbf{X} is given by

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1} \mathbf{X}^\top \mathbf{X}. \quad (5.85)$$

PCA finds a representation (through linear transformation) $\mathbf{z} = \mathbf{W}^\top \mathbf{x}$, where $\text{Var}[\mathbf{z}]$ is diagonal.

In section 2.12, we saw that the principal components of a design matrix \mathbf{X} are given by the eigenvectors of $\mathbf{X}^\top \mathbf{X}$. From this view,

$$\mathbf{X}^\top \mathbf{X} = \mathbf{W} \Lambda \mathbf{W}^\top. \quad (5.86)$$

In this section, we exploit an alternative derivation of the principal components. The principal components may also be obtained via singular value decomposition (SVD). Specifically, they are the right singular vectors of \mathbf{X} . To see this, let \mathbf{W} be the right singular vectors in the decomposition $\mathbf{X} = \mathbf{U} \Sigma \mathbf{W}^\top$. We then recover the original eigenvector equation with \mathbf{W} as the eigenvector basis:

$$\mathbf{X}^\top \mathbf{X} = (\mathbf{U} \Sigma \mathbf{W}^\top)^\top \mathbf{U} \Sigma \mathbf{W}^\top = \mathbf{W} \Sigma^2 \mathbf{W}^\top. \quad (5.87)$$

The SVD is helpful to show that PCA results in a diagonal $\text{Var}[\mathbf{z}]$. Using the SVD of \mathbf{X} , we can express the variance of \mathbf{X} as:

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1} \mathbf{X}^\top \mathbf{X} \quad (5.88)$$

$$= \frac{1}{m-1} (\mathbf{U} \Sigma \mathbf{W}^\top)^\top \mathbf{U} \Sigma \mathbf{W}^\top \quad (5.89)$$

$$= \frac{1}{m-1} \mathbf{W} \Sigma^\top \mathbf{U}^\top \mathbf{U} \Sigma \mathbf{W}^\top \quad (5.90)$$

$$= \frac{1}{m-1} \mathbf{W} \Sigma^2 \mathbf{W}^\top, \quad (5.91)$$

where we use the fact that $\mathbf{U}^\top \mathbf{U} = \mathbf{I}$ because the \mathbf{U} matrix of the singular value decomposition is defined to be orthogonal. This shows that the covariance of \mathbf{z} is diagonal as required:

$$\text{Var}[\mathbf{z}] = \frac{1}{m-1} \mathbf{Z}^\top \mathbf{Z} \quad (5.92)$$

$$= \frac{1}{m-1} \mathbf{W}^\top \mathbf{X}^\top \mathbf{X} \mathbf{W} \quad (5.93)$$

$$= \frac{1}{m-1} \mathbf{W}^\top \mathbf{W} \Sigma^2 \mathbf{W}^\top \mathbf{W} \quad (5.94)$$

$$= \frac{1}{m-1} \Sigma^2, \quad (5.95)$$

where this time we use the fact that $\mathbf{W}^\top \mathbf{W} = \mathbf{I}$, again from the definition of the SVD.

The above analysis shows that when we project the data \mathbf{x} to \mathbf{z} , via the linear transformation \mathbf{W} , the resulting representation has a diagonal covariance matrix (as given by Σ^2), which immediately implies that the individual elements of \mathbf{z} are mutually uncorrelated.

This ability of PCA to transform data into a representation where the elements are mutually uncorrelated is a very important property of PCA. It is a simple example of a representation that attempts to *disentangle the unknown factors of variation* underlying the data. In the case of PCA, this disentangling takes the form of finding a rotation of the input space (described by \mathbf{W}) that aligns the principal axes of variance with the basis of the new representation space associated with \mathbf{z} .

While correlation is an important category of dependency between elements of the data, we are also interested in learning representations that disentangle more complicated forms of feature dependencies. For this, we will need more than what can be done with a simple linear transformation.

5.8.2 k -means Clustering

Another example of a simple representation learning algorithm is k -means clustering. The k -means clustering algorithm divides the training set into k different clusters of examples that are near each other. We can thus think of the algorithm as providing a k -dimensional one-hot code vector \mathbf{h} representing an input \mathbf{x} . If \mathbf{x} belongs to cluster i , then $h_i = 1$, and all other entries of the representation \mathbf{h} are zero.

The one-hot code provided by k -means clustering is an example of a sparse representation, because the majority of its entries are zero for every input. Later, we develop other algorithms that learn more flexible sparse representations, where more than one entry can be nonzero for each input \mathbf{x} . One-hot codes are an extreme example of sparse representations that lose many of the benefits of a distributed representation. The one-hot code still confers some statistical advantages (it naturally conveys the idea that all examples in the same cluster are similar to each other), and it confers the computational advantage that the entire representation

may be captured by a single integer.

The k -means algorithm works by initializing k different centroids $\{\boldsymbol{\mu}^{(1)}, \dots, \boldsymbol{\mu}^{(k)}\}$ to different values, then alternating between two different steps until convergence. In one step, each training example is assigned to cluster i , where i is the index of the nearest centroid $\boldsymbol{\mu}^{(i)}$. In the other step, each centroid $\boldsymbol{\mu}^{(i)}$ is updated to the mean of all training examples $\mathbf{x}^{(j)}$ assigned to cluster i .

One difficulty pertaining to clustering is that the clustering problem is inherently ill posed, in the sense that there is no single criterion that measures how well a clustering of the data corresponds to the real world. We can measure properties of the clustering, such as the average Euclidean distance from a cluster centroid to the members of the cluster. This enables us to tell how well we are able to reconstruct the training data from the cluster assignments. We do not know how well the cluster assignments correspond to properties of the real world. Moreover, there may be many different clusterings that all correspond well to some property of the real world. We may hope to find a clustering that relates to one feature but obtain a different, equally valid clustering that is not relevant to our task. For example, suppose that we run two clustering algorithms on a dataset consisting of images of red trucks, images of red cars, images of gray trucks, and images of gray cars. If we ask each clustering algorithm to find two clusters, one algorithm may find a cluster of cars and a cluster of trucks, while another may find a cluster of red vehicles and a cluster of gray vehicles. Suppose we also run a third clustering algorithm, which is allowed to determine the number of clusters. This may assign the examples to four clusters, red cars, red trucks, gray cars, and gray trucks. This new clustering now at least captures information about both attributes, but it has lost information about similarity. Red cars are in a different cluster from gray cars, just as they are in a different cluster from gray trucks. The output of the clustering algorithm does not tell us that red cars are more similar to gray cars than they are to gray trucks. They are different from both things, and that is all we know.

These issues illustrate some of the reasons that we may prefer a distributed representation to a one-hot representation. A distributed representation could have two attributes for each vehicle—one representing its color and one representing whether it is a car or a truck. It is still not entirely clear what the optimal distributed representation is (how can the learning algorithm know whether the two attributes we are interested in are color and car-versus-truck rather than manufacturer and age?), but having many attributes reduces the burden on the algorithm to guess which single attribute we care about, and gives us the ability to measure similarity between objects in a fine-grained way by comparing many

attributes instead of just testing whether one attribute matches.

5.9 Stochastic Gradient Descent

Nearly all of deep learning is powered by one very important algorithm: **stochastic gradient descent** (SGD). Stochastic gradient descent is an extension of the gradient descent algorithm introduced in section 4.3.

A recurring problem in machine learning is that large training sets are necessary for good generalization, but large training sets are also more computationally expensive.

The cost function used by a machine learning algorithm often decomposes as a sum over training examples of some per-example loss function. For example, the negative conditional log-likelihood of the training data can be written as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \quad (5.96)$$

where L is the per-example loss $L(\mathbf{x}, y, \boldsymbol{\theta}) = -\log p(y | \mathbf{x}; \boldsymbol{\theta})$.

For these additive cost functions, gradient descent requires computing

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \quad (5.97)$$

The computational cost of this operation is $O(m)$. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long.

The insight of SGD is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each step of the algorithm, we can sample a **minibatch** of examples $\mathbb{B} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m')}\}$ drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples, ranging from one to a few hundred. Crucially, m' is usually held fixed as the training set size m grows. We may fit a training set with billions of examples using updates computed on only a hundred examples.

The estimate of the gradient is formed as

$$\mathbf{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \quad (5.98)$$

using examples from the minibatch \mathbb{B} . The stochastic gradient descent algorithm then follows the estimated gradient downhill:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{g}, \quad (5.99)$$

where ϵ is the learning rate.

Gradient descent in general has often been regarded as slow or unreliable. In the past, the application of gradient descent to nonconvex optimization problems was regarded as foolhardy or unprincipled. Today, we know that the machine learning models described in part II work very well when trained with gradient descent. The optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function quickly enough to be useful.

Stochastic gradient descent has many important uses outside the context of deep learning. It is the main way to train large linear models on very large datasets. For a fixed model size, the cost per SGD update does not depend on the training set size m . In practice, we often use a larger model as the training set size increases, but we are not forced to do so. The number of updates required to reach convergence usually increases with training set size. However, as m approaches infinity, the model will eventually converge to its best possible test error before SGD has sampled every example in the training set. Increasing m further will not extend the amount of training time needed to reach the model's best possible test error. From this point of view, one can argue that the asymptotic cost of training a model with SGD is $O(1)$ as a function of m .

Prior to the advent of deep learning, the main way to learn nonlinear models was to use the kernel trick in combination with a linear model. Many kernel learning algorithms require constructing an $m \times m$ matrix $G_{i,j} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. Constructing this matrix has computational cost $O(m^2)$, which is clearly undesirable for datasets with billions of examples. In academia, starting in 2006, deep learning was initially interesting because it was able to generalize to new examples better than competing algorithms when trained on medium-sized datasets with tens of thousands of examples. Soon after, deep learning garnered additional interest in industry because it provided a scalable way of training nonlinear models on large datasets.

Stochastic gradient descent and many enhancements to it are described further in chapter 8.

5.10 Building a Machine Learning Algorithm

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe: combine a specification of a dataset, a cost function, an optimization procedure and a model.

For example, the linear regression algorithm combines a dataset consisting of \mathbf{X} and \mathbf{y} , the cost function

$$J(\mathbf{w}, b) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}), \quad (5.100)$$

the model specification $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; \mathbf{x}^\top \mathbf{w} + b, 1)$, and, in most cases, the optimization algorithm defined by solving for where the gradient of the cost is zero using the normal equations.

By realizing that we can replace any of these components mostly independently from the others, we can obtain a wide range of algorithms.

The cost function typically includes at least one term that causes the learning process to perform statistical estimation. The most common cost function is the negative log-likelihood, so that minimizing the cost function causes maximum likelihood estimation.

The cost function may also include additional terms, such as regularization terms. For example, we can add weight decay to the linear regression cost function to obtain

$$J(\mathbf{w}, b) = \lambda \|\mathbf{w}\|_2^2 - \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}). \quad (5.101)$$

This still allows closed form optimization.

If we change the model to be nonlinear, then most cost functions can no longer be optimized in closed form. This requires us to choose an iterative numerical optimization procedure, such as gradient descent.

The recipe for constructing a learning algorithm by combining models, costs, and optimization algorithms supports both supervised and unsupervised learning. The linear regression example shows how to support supervised learning. Unsupervised learning can be supported by defining a dataset that contains only \mathbf{X} and providing an appropriate unsupervised cost and model. For example, we can obtain the first PCA vector by specifying that our loss function is

$$J(\mathbf{w}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \|\mathbf{x} - r(\mathbf{x}; \mathbf{w})\|_2^2 \quad (5.102)$$

while our model is defined to have \mathbf{w} with norm one and reconstruction function $r(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} \mathbf{w}$.

In some cases, the cost function may be a function that we cannot actually evaluate, for computational reasons. In these cases, we can still approximately minimize it using iterative numerical optimization, as long as we have some way of approximating its gradients.

Most machine learning algorithms make use of this recipe, though it may not be immediately obvious. If a machine learning algorithm seems especially unique or hand designed, it can usually be understood as using a special-case optimizer. Some models, such as decision trees and k -means, require special-case optimizers because their cost functions have flat regions that make them inappropriate for minimization by gradient-based optimizers. Recognizing that most machine learning algorithms can be described using this recipe helps to see the different algorithms as part of a taxonomy of methods for doing related tasks that work for similar reasons, rather than as a long list of algorithms that each have separate justifications.

5.11 Challenges Motivating Deep Learning

The simple machine learning algorithms described in this chapter work well on a wide variety of important problems. They have not succeeded, however, in solving the central problems in AI, such as recognizing speech or recognizing objects.

The development of deep learning was motivated in part by the failure of traditional algorithms to generalize well on such AI tasks.

This section is about how the challenge of generalizing to new examples becomes exponentially more difficult when working with high-dimensional data, and how the mechanisms used to achieve generalization in traditional machine learning are insufficient to learn complicated functions in high-dimensional spaces. Such spaces also often impose high computational costs. Deep learning was designed to overcome these and other obstacles.

5.11.1 The Curse of Dimensionality

Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high. This phenomenon is known as the **curse of dimensionality**. Of particular concern is that the number of possible distinct configurations of a set of variables increases exponentially as the number of variables increases.

The curse of dimensionality arises in many places in computer science, especially in machine learning.



Figure 5.9: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. (*Left*) In this one-dimensional example, we have one variable for which we only care to distinguish 10 regions of interest. With enough examples falling within each of these regions (each region corresponds to a cell in the illustration), learning algorithms can easily generalize correctly. A straightforward way to generalize is to estimate the value of the target function within each region (and possibly interpolate between neighboring regions). (*Center*) With two dimensions, it is more difficult to distinguish 10 different values of each variable. We need to keep track of up to $10 \times 10 = 100$ regions, and we need at least that many examples to cover all those regions. (*Right*) With three dimensions, this grows to $10^3 = 1,000$ regions and at least that many examples. For d dimensions and v values to be distinguished along each axis, we seem to need $O(v^d)$ regions and examples. This is an instance of the curse of dimensionality. Figure graciously provided by Nicolas Chapados.

One challenge posed by the curse of dimensionality is a statistical challenge. As illustrated in figure 5.9, a statistical challenge arises because the number of possible configurations of \mathbf{x} is much larger than the number of training examples. To understand the issue, let us consider that the input space is organized into a grid, as in the figure. We can describe low-dimensional space with a small number of grid cells that are mostly occupied by the data. When generalizing to a new data point, we can usually tell what to do simply by inspecting the training examples that lie in the same cell as the new input. For example, if estimating the probability density at some point \mathbf{x} , we can just return the number of training examples in the same unit volume cell as \mathbf{x} , divided by the total number of training examples. If we wish to classify an example, we can return the most common class of training examples in the same cell. If we are doing regression, we can average the target values observed over the examples in that cell. But what about the cells for which we have seen no example? Because in high-dimensional spaces, the number of configurations is huge, much larger than our number of examples, a typical grid cell has no training example associated with it. How could we possibly say something meaningful about these new configurations? Many traditional machine learning

algorithms simply assume that the output at a new point should be approximately the same as the output at the nearest training point.

5.11.2 Local Constancy and Smoothness Regularization

To generalize well, machine learning algorithms need to be guided by prior beliefs about what kind of function they should learn. We have seen these priors incorporated as explicit beliefs in the form of probability distributions over parameters of the model. More informally, we may also discuss prior beliefs as directly influencing the *function* itself and influencing the parameters only indirectly, as a result of the relationship between the parameters and the function. Additionally, we informally discuss prior beliefs as being expressed implicitly by choosing algorithms that are biased toward choosing some class of functions over another, even though these biases may not be expressed (or even be possible to express) in terms of a probability distribution representing our degree of belief in various functions.

Among the most widely used of these implicit “priors” is the **smoothness prior**, or **local constancy prior**. This prior states that the function we learn should not change very much within a small region.

Many simpler algorithms rely exclusively on this prior to generalize well, and as a result, they fail to scale to the statistical challenges involved in solving AI-level tasks. Throughout this book, we describe how deep learning introduces additional (explicit and implicit) priors in order to reduce the generalization error on sophisticated tasks. Here, we explain why the smoothness prior alone is insufficient for these tasks.

There are many different ways to implicitly or explicitly express a prior belief that the learned function should be smooth or locally constant. All these different methods are designed to encourage the learning process to learn a function f^* that satisfies the condition

$$f^*(\mathbf{x}) \approx f^*(\mathbf{x} + \epsilon) \quad (5.103)$$

for most configurations \mathbf{x} and small change ϵ . In other words, if we know a good answer for an input \mathbf{x} (for example, if \mathbf{x} is a labeled training example), then that answer is probably good in the neighborhood of \mathbf{x} . If we have several good answers in some neighborhood, we would combine them (by some form of averaging or interpolation) to produce an answer that agrees with as many of them as much as possible.

An extreme example of the local constancy approach is the k -nearest neighbors family of learning algorithms. These predictors are literally constant over each region containing all the points \mathbf{x} that have the same set of k nearest neighbors in

the training set. For $k = 1$, the number of distinguishable regions cannot be more than the number of training examples.

While the k -nearest neighbors algorithm copies the output from nearby training examples, most kernel machines interpolate between training set outputs associated with nearby training examples. An important class of kernels is the family of **local kernels**, where $k(\mathbf{u}, \mathbf{v})$ is large when $\mathbf{u} = \mathbf{v}$ and decreases as \mathbf{u} and \mathbf{v} grow further apart from each other. A local kernel can be thought of as a similarity function that performs template matching, by measuring how closely a test example \mathbf{x} resembles each training example $\mathbf{x}^{(i)}$. Much of the modern motivation for deep learning is derived from studying the limitations of local template matching and how deep models are able to succeed in cases where local template matching fails (Bengio *et al.*, 2006b).

Decision trees also suffer from the limitations of exclusively smoothness-based learning, because they break the input space into as many regions as there are leaves and use a separate parameter (or sometimes many parameters for extensions of decision trees) in each region. If the target function requires a tree with at least n leaves to be represented accurately, then at least n training examples are required to fit the tree. A multiple of n is needed to achieve some level of statistical confidence in the predicted output.

In general, to distinguish $O(k)$ regions in input space, all these methods require $O(k)$ examples. Typically there are $O(k)$ parameters, with $O(1)$ parameters associated with each of the $O(k)$ regions. The nearest neighbor scenario, in which each training example can be used to define at most one region, is illustrated in figure 5.10.

Is there a way to represent a complex function that has many more regions to be distinguished than the number of training examples? Clearly, assuming only smoothness of the underlying function will not allow a learner to do that. For example, imagine that the target function is a kind of checkerboard. A checkerboard contains many variations, but there is a simple structure to them. Imagine what happens when the number of training examples is substantially smaller than the number of black and white squares on the checkerboard. Based on only local generalization and the smoothness or local constancy prior, the learner would be guaranteed to correctly guess the color of a new point if it lay within the same checkerboard square as a training example. There is no guarantee, however, that the learner could correctly extend the checkerboard pattern to points lying in squares that do not contain training examples. With this prior alone, the only information that an example tells us is the color of its square, and the only way to get the colors of the entire checkerboard right is to cover each of its cells with at

least one example.

The smoothness assumption and the associated nonparametric learning algorithms work extremely well as long as there are enough examples for the learning algorithm to observe high points on most peaks and low points on most valleys of the true underlying function to be learned. This is generally true when the function to be learned is smooth enough and varies in few enough dimensions. In high dimensions, even a very smooth function can change smoothly but in a different way along each dimension. If the function additionally behaves differently in various regions, it can become extremely complicated to describe with a set of training examples. If the function is complicated (we want to distinguish a huge number of regions compared to the number of examples), is there any hope to generalize well?

The answer to both of these questions—whether it is possible to represent a complicated function efficiently, and whether it is possible for the estimated function to generalize well to new inputs—is yes. The key insight is that a very large number of regions, such as $O(2^k)$, can be defined with $O(k)$ examples, so long as we

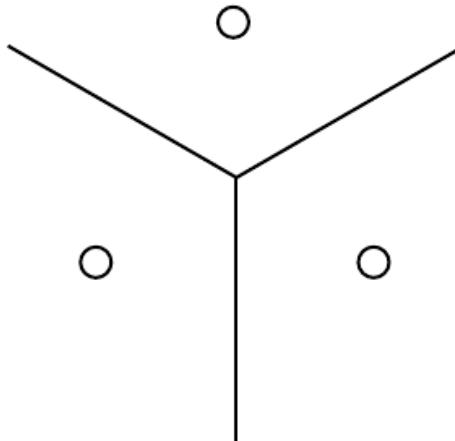


Figure 5.10: Illustration of how the nearest neighbor algorithm breaks up the input space into regions. An example (represented here by a circle) within each region defines the region boundary (represented here by the lines). The y value associated with each example defines what the output should be for all points within the corresponding region. The regions defined by nearest neighbor matching form a geometric pattern called a Voronoi diagram. The number of these contiguous regions cannot grow faster than the number of training examples. While this figure illustrates the behavior of the nearest neighbor algorithm specifically, other machine learning algorithms that rely exclusively on the local smoothness prior for generalization exhibit similar behaviors: each training example only informs the learner about how to generalize in some neighborhood immediately surrounding that example.

introduce some dependencies between the regions through additional assumptions about the underlying data-generating distribution. In this way, we can actually generalize nonlocally (Bengio and Monperrus, 2005; Bengio *et al.*, 2006c). Many different deep learning algorithms provide implicit or explicit assumptions that are reasonable for a broad range of AI tasks in order to capture these advantages.

Other approaches to machine learning often make stronger, task-specific assumptions. For example, we could easily solve the checkerboard task by providing the assumption that the target function is periodic. Usually we do not include such strong, task-specific assumptions in neural networks so that they can generalize to a much wider variety of structures. AI tasks have structure that is much too complex to be limited to simple, manually specified properties such as periodicity, so we want learning algorithms that embody more general-purpose assumptions. The core idea in deep learning is that we assume that the data was generated by the *composition of factors*, or features, potentially at multiple levels in a hierarchy. Many other similarly generic assumptions can further improve deep learning algorithms. These apparently mild assumptions allow an exponential gain in the relationship between the number of examples and the number of regions that can be distinguished. We describe these exponential gains more precisely in sections 6.4.1, 15.4 and 15.5. The exponential advantages conferred by the use of deep distributed representations counter the exponential challenges posed by the curse of dimensionality.

5.11.3 Manifold Learning

An important concept underlying many ideas in machine learning is that of a manifold.

A **manifold** is a connected region. Mathematically, it is a set of points associated with a neighborhood around each point. From any given point, the manifold locally appears to be a Euclidean space. In everyday life, we experience the surface of the world as a 2-D plane, but it is in fact a spherical manifold in 3-D space.

The concept of a neighborhood surrounding each point implies the existence of transformations that can be applied to move on the manifold from one position to a neighboring one. In the example of the world’s surface as a manifold, one can walk north, south, east, or west.

Although there is a formal mathematical meaning to the term “manifold,” in machine learning it tends to be used more loosely to designate a connected set of points that can be approximated well by considering only a small number of

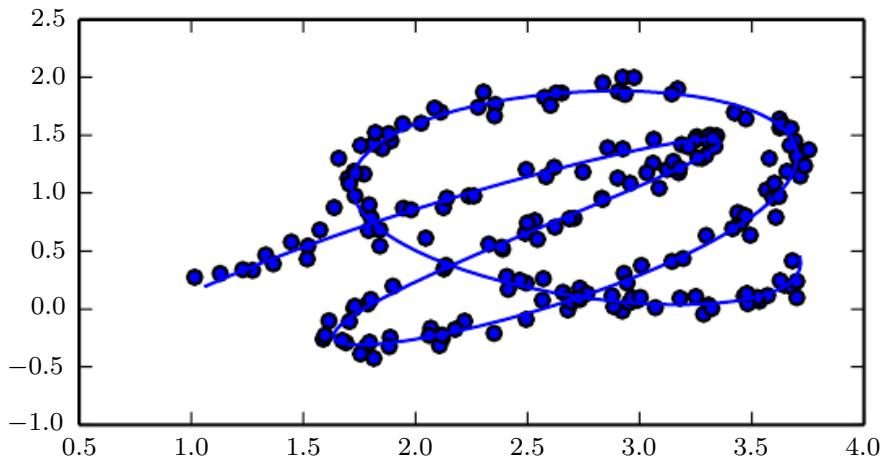


Figure 5.11: Data sampled from a distribution in a two-dimensional space that is actually concentrated near a one-dimensional manifold, like a twisted string. The solid line indicates the underlying manifold that the learner should infer.

degrees of freedom, or dimensions, embedded in a higher-dimensional space. Each dimension corresponds to a local direction of variation. See figure 5.11 for an example of training data lying near a one-dimensional manifold embedded in two-dimensional space. In the context of machine learning, we allow the dimensionality of the manifold to vary from one point to another. This often happens when a manifold intersects itself. For example, a figure eight is a manifold that has a single dimension in most places but two dimensions at the intersection at the center.

Many machine learning problems seem hopeless if we expect the machine learning algorithm to learn functions with interesting variations across all of \mathbb{R}^n . **Manifold learning** algorithms surmount this obstacle by assuming that most of \mathbb{R}^n consists of invalid inputs, and that interesting inputs occur only along a collection of manifolds containing a small subset of points, with interesting variations in the output of the learned function occurring only along directions that lie on the manifold, or with interesting variations happening only when we move from one manifold to another. Manifold learning was introduced in the case of continuous-valued data and in the unsupervised learning setting, although this probability concentration idea can be generalized to both discrete data and the supervised learning setting: the key assumption remains that probability mass is highly concentrated.

The assumption that the data lies along a low-dimensional manifold may not always be correct or useful. We argue that in the context of AI tasks, such as those that involve processing images, sounds, or text, the manifold assumption is

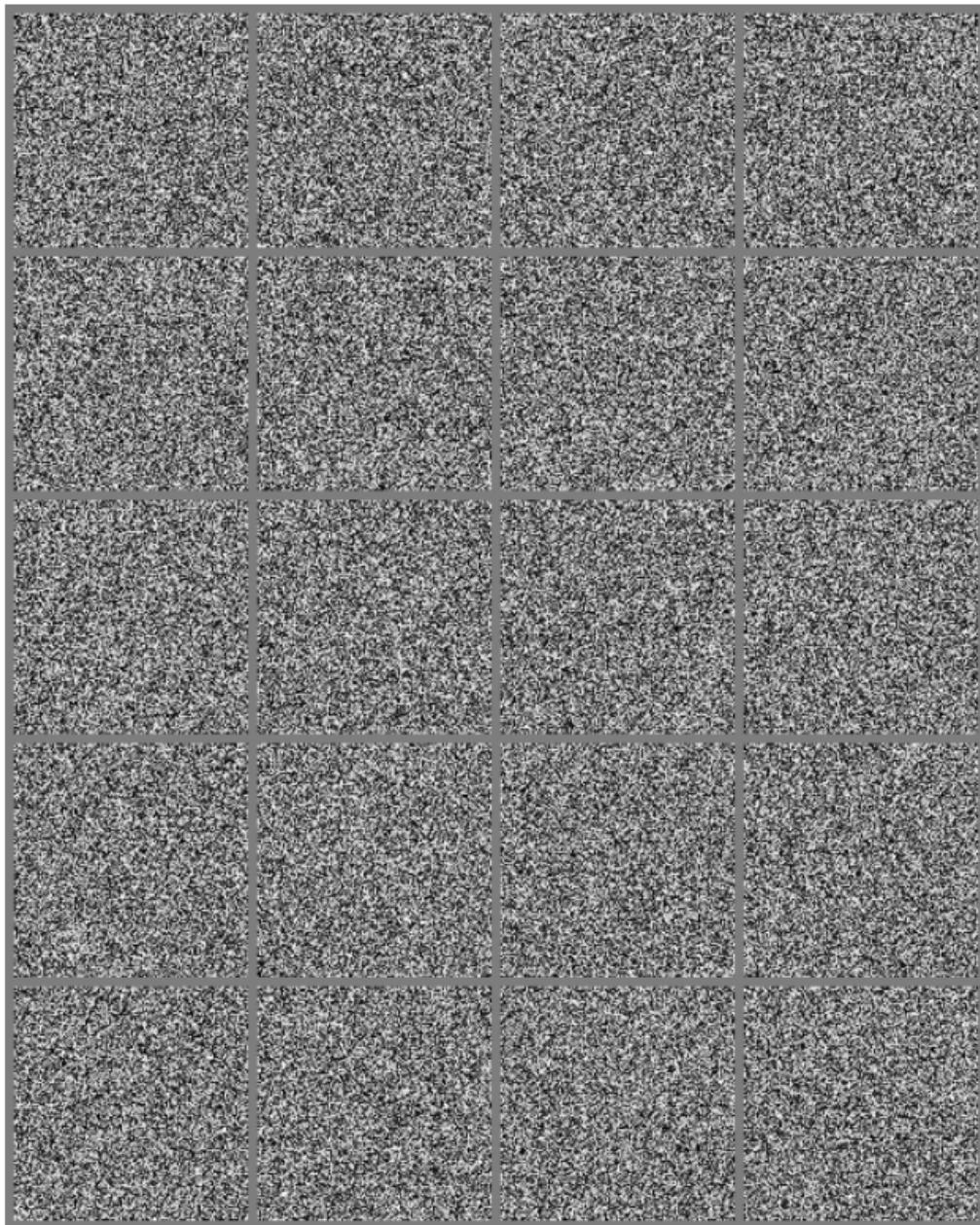


Figure 5.12: Sampling images uniformly at random (by randomly picking each pixel according to a uniform distribution) gives rise to noisy images. Although there is a nonzero probability of generating an image of a face or of any other object frequently encountered in AI applications, we never actually observe this happening in practice. This suggests that the images encountered in AI applications occupy a negligible proportion of the volume of image space.

at least approximately correct. The evidence in favor of this assumption consists of two categories of observations.

The first observation in favor of the **manifold hypothesis** is that the proba-

bility distribution over images, text strings, and sounds that occur in real life is highly concentrated. Uniform noise essentially never resembles structured inputs from these domains. Figure 5.12 shows how, instead, uniformly sampled points look like the patterns of static that appear on analog television sets when no signal is available. Similarly, if you generate a document by picking letters uniformly at random, what is the probability that you will get a meaningful English-language text? Almost zero, again, because most of the long sequences of letters do not correspond to a natural language sequence: the distribution of natural language sequences occupies a very little volume in the total space of sequences of letters.

Of course, concentrated probability distributions are not sufficient to show that the data lies on a reasonably small number of manifolds. We must also establish that the examples we encounter are connected to each other by other examples, with each example surrounded by other highly similar examples that can be reached by applying transformations to traverse the manifold. The second argument in favor of the manifold hypothesis is that we can imagine such neighborhoods and transformations, at least informally. In the case of images, we can certainly think of many possible transformations that allow us to trace out a manifold in image space: we can gradually dim or brighten the lights, gradually move or rotate objects in the image, gradually alter the colors on the surfaces of objects, and so forth. Multiple manifolds are likely involved in most applications. For example, the manifold of human face images may not be connected to the manifold of cat face images.

These thought experiments convey some intuitive reasons supporting the manifold hypothesis. More rigorous experiments (Cayton, 2005; Narayanan and Mitter, 2010; Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004) clearly support the hypothesis for a large class of datasets of interest in AI.

When the data lies on a low-dimensional manifold, it can be most natural for machine learning algorithms to represent the data in terms of coordinates on the manifold, rather than in terms of coordinates in \mathbb{R}^n . In everyday life, we can think of roads as 1-D manifolds embedded in 3-D space. We give directions to specific addresses in terms of address numbers along these 1-D roads, not in terms of coordinates in 3-D space. Extracting these manifold coordinates is challenging but holds the promise of improving many machine learning algorithms. This general principle is applied in many contexts. Figure 5.13 shows the manifold structure of a dataset consisting of faces. By the end of this book, we will have developed the methods necessary to learn such a manifold structure. In figure 20.6, we will see how a machine learning algorithm can successfully accomplish this goal.

This concludes part I, which has provided the basic concepts in mathematics and machine learning that are employed throughout the remaining parts of the book. You are now prepared to embark on your study of deep learning.



Figure 5.13: Training examples from the QMUL Multiview Face Dataset ([Gong et al., 2000](#)), for which the subjects were asked to move in such a way as to cover the two-dimensional manifold corresponding to two angles of rotation. We would like learning algorithms to be able to discover and disentangle such manifold coordinates. Figure 20.6 illustrates such a feat.

Chapter 6

Deep Feedforward Networks

Deep feedforward networks, also called **feedforward neural networks**, or **multilayer perceptrons** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^* . For example, for a classifier, $y = f^*(\mathbf{x})$ maps an input \mathbf{x} to a category y . A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

These models are called **feedforward** because information flows through the function being evaluated from \mathbf{x} , through the intermediate computations used to define f , and finally to the output \mathbf{y} . There are no **feedback** connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**, as presented in chapter 10.

Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications. For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the **second layer**, and so on. The

overall length of the chain gives the **depth** of the model. The name “deep learning” arose from this terminology. The final layer of a feedforward network is called the **output layer**. During neural network training, we drive $f(\mathbf{x})$ to match $f^*(\mathbf{x})$. The training data provides us with noisy, approximate examples of $f^*(\mathbf{x})$ evaluated at different training points. Each example \mathbf{x} is accompanied by a label $y \approx f^*(\mathbf{x})$. The training examples specify directly what the output layer must do at each point \mathbf{x} ; it must produce a value that is close to y . The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^* . Because the training data does not show the desired output for each of these layers, they are called **hidden layers**.

Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representations is drawn from neuroscience. The choice of the functions $f^{(i)}(\mathbf{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. Modern neural network research, however, is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they can be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

To extend linear models to represent nonlinear functions of \mathbf{x} , we can apply the linear model not to \mathbf{x} itself but to a transformed input $\phi(\mathbf{x})$, where ϕ is a

nonlinear transformation. Equivalently, we can apply the kernel trick described in section 5.7.2, to obtain a nonlinear learning algorithm based on implicitly applying the ϕ mapping. We can think of ϕ as providing a set of features describing \mathbf{x} , or as providing a new representation for \mathbf{x} .

The question is then how to choose the mapping ϕ .

1. One option is to use a very generic ϕ , such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel. If $\phi(\mathbf{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor. Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.
2. Another option is to manually engineer ϕ . Until the advent of deep learning, this was the dominant approach. It requires decades of human effort for each separate task, with practitioners specializing in different domains, such as speech recognition or computer vision, and with little transfer between domains.
3. The strategy of deep learning is to learn ϕ . In this approach, we have a model $y = f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \phi(\mathbf{x}; \boldsymbol{\theta})^\top \mathbf{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters \mathbf{w} that map from $\phi(\mathbf{x})$ to the desired output. This is an example of a deep feedforward network, with ϕ defining a hidden layer. This approach is the only one of the three that gives up on the convexity of the training problem, but the benefits outweigh the harms. In this approach, we parametrize the representation as $\phi(\mathbf{x}; \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation. If we wish, this approach can capture the benefit of the first approach by being highly generic—we do so by using a very broad family $\phi(\mathbf{x}; \boldsymbol{\theta})$. Deep learning can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(\mathbf{x}; \boldsymbol{\theta})$ that they expect will perform well. The advantage is that the human designer only needs to find the right general function family rather than finding precisely the right function.

This general principle of improving models by learning features extends beyond the feedforward networks described in this chapter. It is a recurring theme of deep learning that applies to all the kinds of models described throughout this book. Feedforward networks are the application of this principle to learning

deterministic mappings from \mathbf{x} to \mathbf{y} that lack feedback connections. Other models, presented later, apply these principles to learning stochastic mappings, functions with feedback, and probability distributions over a single vector.

We begin this chapter with a simple example of a feedforward network. Next, we address each of the design decisions needed to deploy a feedforward network. First, training a feedforward network requires making many of the same design decisions as are necessary for a linear model: choosing the optimizer, the cost function, and the form of the output units. We review these basics of gradient-based learning, then proceed to confront some of the design decisions that are unique to feedforward networks. Feedforward networks have introduced the concept of a hidden layer, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values. We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer. Learning in deep neural networks requires computing the gradients of complicated functions. We present the **back-propagation** algorithm and its modern generalizations, which can be used to efficiently compute these gradients. Finally, we close with some historical perspective.

6.1 Example: Learning XOR

To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

The XOR function (“exclusive or”) is an operation on two binary values, x_1 and x_2 . When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(\mathbf{x})$ that we want to learn. Our model provides a function $y = f(\mathbf{x}; \boldsymbol{\theta})$, and our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^* .

In this simple example, we will not be concerned with statistical generalization. We want our network to perform correctly on the four points $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top, [1, 1]^\top\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

We can treat this problem as a regression problem and use a mean squared error loss function. We have chosen this loss function to simplify the math for this example as much as possible. In practical applications, MSE is usually not an

appropriate cost function for modeling binary data. More appropriate approaches are described in section 6.2.2.2.

Evaluated on our whole training set, the MSE loss function is

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\mathbf{x} \in \mathbb{X}} (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2. \quad (6.1)$$

Now we must choose the form of our model, $f(\mathbf{x}; \boldsymbol{\theta})$. Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of \mathbf{w} and b . Our model is defined to be

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b. \quad (6.2)$$

We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to \mathbf{w} and b using the normal equations.

After solving the normal equations, we obtain $\mathbf{w} = \mathbf{0}$ and $b = \frac{1}{2}$. The linear model simply outputs 0.5 everywhere. Why does this happen? Figure 6.1 shows how a linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

Specifically, we will introduce a simple feedforward network with one hidden layer containing two hidden units. See figure 6.2 for an illustration of this model. This feedforward network has a vector of hidden units \mathbf{h} that are computed by a function $f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to \mathbf{h} rather than to \mathbf{x} . The network now contains two functions chained together, $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$ and $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$, with the complete model being $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$.

What function should $f^{(1)}$ compute? Linear models have served us well so far, and it may be tempting to make $f^{(1)}$ linear as well. Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input. Ignoring the intercept terms for the moment, suppose $f^{(1)}(\mathbf{x}) = \mathbf{W}^\top \mathbf{x}$ and $f^{(2)}(\mathbf{h}) = \mathbf{h}^\top \mathbf{w}$. Then $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{W} \mathbf{w}$. We could represent this function as $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}'$ where $\mathbf{w}' = \mathbf{W} \mathbf{w}$.

Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed nonlinear function called an activation function. We use that strategy here, by defining $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{c})$, where \mathbf{W} provides the weights of a linear transformation and \mathbf{c} the biases. Previously, to describe a linear regression model, we used a vector of weights and a scalar bias parameter to describe an

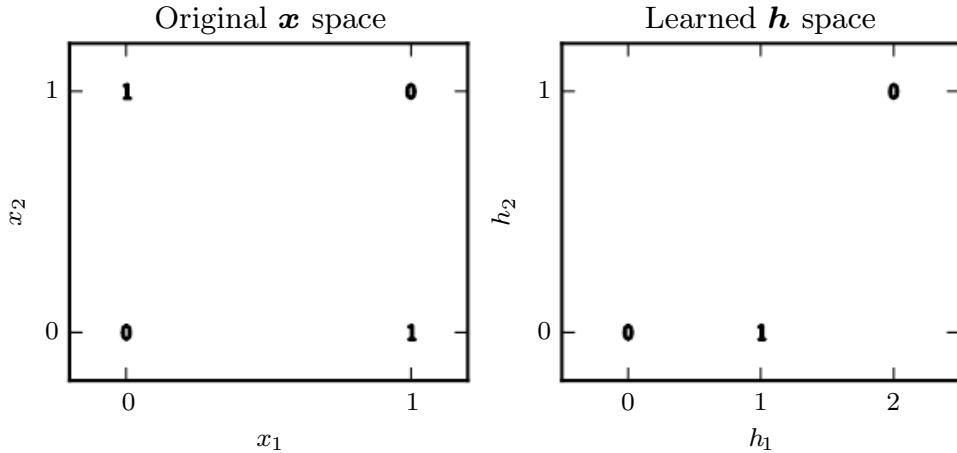


Figure 6.1: Solving the XOR problem by learning a representation. The bold numbers printed on the plot indicate the value that the learned function must output at each point. (*Left*) A linear model applied directly to the original input cannot implement the XOR function. When $x_1 = 0$, the model's output must increase as x_2 increases. When $x_1 = 1$, the model's output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2 . The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem. (*Right*) In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the nonlinear features have mapped both $\mathbf{x} = [1, 0]^\top$ and $\mathbf{x} = [0, 1]^\top$ to a single point in feature space, $\mathbf{h} = [1, 0]^\top$. The linear model can now describe the function as increasing in h_1 and decreasing in h_2 . In this example, the motivation for learning the feature space is only to make the model capacity greater so that it can fit the training set. In more realistic applications, learned representations can also help the model to generalize.

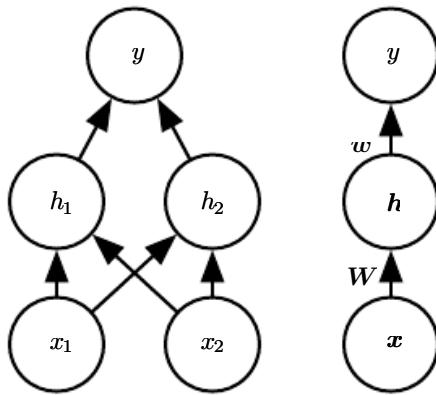


Figure 6.2: An example of a feedforward network, drawn in two different styles. Specifically, this is the feedforward network we use to solve the XOR example. It has a single hidden layer containing two units. (*Left*) In this style, we draw every unit as a node in the graph. This style is explicit and unambiguous, but for networks larger than this example, it can consume too much space. (*Right*) In this style, we draw a node in the graph for each entire vector representing a layer’s activations. This style is much more compact. Sometimes we annotate the edges in this graph with the name of the parameters that describe the relationship between two layers. Here, we indicate that a matrix \mathbf{W} describes the mapping from \mathbf{x} to \mathbf{h} , and a vector \mathbf{w} describes the mapping from \mathbf{h} to y . We typically omit the intercept parameters associated with each layer when labeling this kind of drawing.

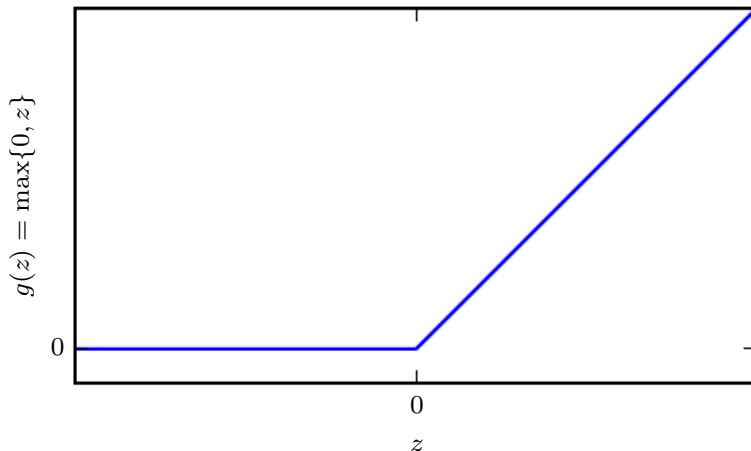


Figure 6.3: The rectified linear activation function. This activation function is the default activation function recommended for use with most feedforward neural networks. Applying this function to the output of a linear transformation yields a nonlinear transformation. The function remains very close to linear, however, in the sense that it is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well. A common principle throughout computer science is that we can build complicated systems from minimal components. Much as a Turing machine’s memory needs only to be able to store 0 or 1 states, we can build a universal function approximator from rectified linear functions.

affine transformation from an input vector to an output scalar. Now, we describe an affine transformation from a vector \mathbf{x} to a vector \mathbf{h} , so an entire vector of bias parameters is needed. The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i)$. In modern neural networks, the default recommendation is to use the **rectified linear unit**, or ReLU ([Jarrett et al., 2009](#); [Nair and Hinton, 2010](#); [Glorot et al., 2011a](#)), defined by the activation function $g(z) = \max\{0, z\}$, depicted in figure 6.3.

We can now specify our complete network as

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b. \quad (6.3)$$

We can then specify a solution to the XOR problem. Let

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad (6.4)$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad (6.5)$$

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad (6.6)$$

and $b = 0$.

We can now walk through how the model processes a batch of inputs. Let \mathbf{X} be the design matrix containing all four points in the binary input space, with one example per row:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \quad (6.7)$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \quad (6.8)$$

Next, we add the bias vector \mathbf{c} , to obtain

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.9)$$

In this space, all the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function. To finish computing the value of \mathbf{h} for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.10)$$

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.

We finish with multiplying by the weight vector \mathbf{w} :

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (6.11)$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here. Instead, a gradient-based optimization algorithm can find parameters that produce very little error. The solution we described to the XOR problem is at a global minimum of the loss function, so gradient descent could converge to this point. There are other equivalent solutions to the XOR problem that gradient descent could also find. The convergence point of gradient descent depends on the initial values of the parameters. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.

6.2 Gradient-Based Learning

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. In section 5.10, we described how to build a machine learning algorithm by specifying an optimization procedure, a cost function, and a model family.

The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become nonconvex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory—in practice it is robust but can encounter numerical problems). Stochastic gradient descent applied to nonconvex loss functions has no such convergence guarantee and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values. The iterative gradient-based optimization algorithms used to train feedforward networks and almost all other deep models are described in detail in chapter 8, with parameter initialization in particular discussed in section 8.4. For the moment, it suffices to understand that the training algorithm is almost always based on using the gradient to descend the cost function in one way or another. The specific algorithms are improvements and refinements on the ideas of gradient descent, introduced in section 4.3, and, more specifically, are most often improvements of the stochastic gradient descent algorithm, introduced in section 5.9.

We can of course train models such as linear regression and support vector machines with gradient descent too, and in fact this is common when the training set is extremely large. From this point of view, training a neural network is not much different from training any other model. Computing the gradient is slightly more complicated for a neural network but can still be done efficiently and exactly. In Section 6.5 we describe how to obtain the gradient using the back-propagation algorithm and modern generalizations of the back-propagation algorithm.

As with other machine learning models, to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model. We now revisit these design considerations with special emphasis on the neural networks scenario.

6.2.1 Cost Functions

An important aspect of the design of a deep neural network is the choice of the cost function. Fortunately, the cost functions for neural networks are more or less

the same as those for other parametric models, such as linear models.

In most cases, our parametric model defines a distribution $p(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function.

Sometimes, we take a simpler approach, where rather than predicting a complete probability distribution over \mathbf{y} , we merely predict some statistic of \mathbf{y} conditioned on \mathbf{x} . Specialized loss functions enable us to train a predictor of these estimates.

The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term. We have already seen some simple examples of regularization applied to linear models in section 5.2.2. The weight decay approach used for linear models is also directly applicable to deep neural networks and is among the most popular regularization strategies. More advanced regularization strategies for neural networks are described in chapter 7.

6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution. This cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}). \quad (6.12)$$

The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$. The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded. For example, as we saw in section 5.5.1, if $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{I})$, then we recover the mean squared error cost,

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + \text{const}, \quad (6.13)$$

up to a scaling factor of $\frac{1}{2}$ and a term that does not depend on $\boldsymbol{\theta}$. The discarded constant is based on the variance of the Gaussian distribution, which in this case we chose not to parametrize. Previously, we saw that the equivalence between maximum likelihood estimation with an output distribution and minimization of

mean squared error holds for a linear model, but in fact, the equivalence holds regardless of the $f(\mathbf{x}; \boldsymbol{\theta})$ used to predict the mean of the Gaussian.

An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\mathbf{y} | \mathbf{x})$ automatically determines a cost function $\log p(\mathbf{y} | \mathbf{x})$.

One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate (become very flat) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate. The negative log-likelihood helps to avoid this problem for many models. Several output units involve an exp function that can saturate when its argument is very negative. The log function in the negative log-likelihood cost function undoes the exp of some output units. We will discuss the interaction between the cost function and the choice of output unit in section 6.2.2.

Avoid letting gradient become too small, i.e. the cost fct too flat.

One unusual property of the cross-entropy cost used to perform maximum likelihood estimation is that it usually does not have a minimum value when applied to the models commonly used in practice. For discrete output variables, most models are parametrized in such a way that they cannot represent a probability of zero or one, but can come arbitrarily close to doing so. Logistic regression is an example of such a model. For real-valued output variables, if the model can control the density of the output distribution (for example, by learning the variance parameter of a Gaussian output distribution) then it becomes possible to assign extremely high density to the correct training set outputs, resulting in cross-entropy approaching negative infinity. Regularization techniques described in chapter 7 provide several different ways of modifying the learning problem so that the model cannot reap unlimited reward in this way.

6.2.1.2 Learning Conditional Statistics

Instead of learning a full probability distribution $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$, we often want to learn just one conditional statistic of \mathbf{y} given \mathbf{x} .

For example, we may have a predictor $f(\mathbf{x}; \boldsymbol{\theta})$ that we wish to employ to predict the mean of \mathbf{y} .

If we use a sufficiently powerful neural network, we can think of the neural network as being able to represent any function f from a wide class of functions, with this class being limited only by features such as continuity and boundedness

rather than by having a specific parametric form. From this point of view, we can view the cost function as being a **functional** rather than just a function. A functional is a mapping from functions to real numbers. We can thus think of learning as choosing a function rather than merely choosing a set of parameters. We can design our cost functional to have its minimum occur at some specific function we desire. For example, we can design the cost functional to have its minimum lie on the function that maps \mathbf{x} to the expected value of \mathbf{y} given \mathbf{x} . Solving an optimization problem with respect to a function requires a mathematical tool called **calculus of variations**, described in section 19.4.2. It is not necessary to understand calculus of variations to understand the content of this chapter. At the moment, it is only necessary to understand that calculus of variations may be used to derive the following two results.

Our first result derived using calculus of variations is that solving the optimization problem

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|^2 \quad (6.14)$$

yields

$$f^*(\mathbf{x}) = \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\mathbf{y} | \mathbf{x})} [\mathbf{y}], \quad (6.15)$$

so long as this function lies within the class we optimize over. In other words, if we could train on infinitely many samples from the true data generating distribution, minimizing the mean squared error cost function would give a function that predicts the mean of \mathbf{y} for each value of \mathbf{x} .

Different cost functions give different statistics. A second result derived using calculus of variations is that

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|_1 \quad (6.16)$$

yields a function that predicts the *median* value of \mathbf{y} for each \mathbf{x} , as long as such a function may be described by the family of functions we optimize over. This cost function is commonly called **mean absolute error**.

Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization. Some output units that saturate produce very small gradients when combined with these cost functions. This is one reason that the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(\mathbf{y} | \mathbf{x})$.

6.2.2 Output Units

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

Any kind of neural network unit that may be used as an output can also be used as a hidden unit. Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well. We revisit these units with additional detail about their use as hidden units in section 6.3.

Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\mathbf{h} = f(\mathbf{x}; \boldsymbol{\theta})$. The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

6.2.2.1 Linear Units for Gaussian Output Distributions

One simple kind of output unit is based on an affine transformation with no nonlinearity. These are often just called linear units.

Given features \mathbf{h} , a layer of linear output units produces a vector $\hat{\mathbf{y}} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$.

Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(\mathbf{y} | \mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I}). \quad (6.17)$$

Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.

The maximum likelihood framework makes it straightforward to learn the covariance of the Gaussian too, or to make the covariance of the Gaussian be a function of the input. However, the covariance must be constrained to be a positive definite matrix for all inputs. It is difficult to satisfy such constraints with a linear output layer, so typically other output units are used to parametrize the covariance. Approaches to modeling the covariance are described shortly, in section 6.2.2.4.

Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

Many tasks require predicting the value of a binary variable y . Classification problems with two classes can be cast in this form.

The maximum likelihood approach is to define a Bernoulli distribution over y conditioned on \mathbf{x} .

A Bernoulli distribution is defined by just a single number. The neural net needs to predict only $P(y = 1 | \mathbf{x})$. For this number to be a valid probability, it must lie in the interval $[0, 1]$.

Satisfying this constraint requires some careful design effort. Suppose we were to use a linear unit and threshold its value to obtain a valid probability:

$$P(y = 1 | \mathbf{x}) = \max \left\{ 0, \min \left\{ 1, \mathbf{w}^\top \mathbf{h} + b \right\} \right\}. \quad (6.18)$$

This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent. Any time that $\mathbf{w}^\top \mathbf{h} + b$ strayed

outside the unit interval, the gradient of the output of the model with respect to its parameters would be $\mathbf{0}$. A gradient of $\mathbf{0}$ is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Instead, it is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer. This approach is based on using sigmoid output units combined with maximum likelihood.

A sigmoid output unit is defined by

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{h} + b), \quad (6.19)$$

where σ is the logistic sigmoid function described in section 3.10.

We can think of the sigmoid output unit as having two components. First, it uses a linear layer to compute $z = \mathbf{w}^\top \mathbf{h} + b$. Next, it uses the sigmoid activation function to convert z into a probability.

We omit the dependence on \mathbf{x} for the moment to discuss how to define a probability distribution over y using the value z . The sigmoid can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, which does not sum to 1. We can then divide by an appropriate constant to obtain a valid probability distribution. If we begin with the assumption that the unnormalized log probabilities are linear in y and z , we can exponentiate to obtain the unnormalized probabilities. We then normalize to see that this yields a Bernoulli distribution controlled by a sigmoidal transformation of z :

$$\log \tilde{P}(y) = yz, \quad (6.20)$$

$$\tilde{P}(y) = \exp(yz), \quad (6.21)$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^1 \exp(y'z)}, \quad (6.22)$$

$$P(y) = \sigma((2y - 1)z). \quad (6.23)$$

Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature. The z variable defining such a distribution over binary variables is called a **logit**.

This approach to predicting the probabilities in log space is natural to use with maximum likelihood learning. Because the cost function used with maximum likelihood is $-\log P(y | \mathbf{x})$, the log in the cost function undoes the exp of the sigmoid. Without this effect, the saturation of the sigmoid could prevent gradient-based learning from making good progress. The loss function for maximum

likelihood learning of a Bernoulli parametrized by a sigmoid is

$$J(\boldsymbol{\theta}) = -\log P(y \mid \mathbf{x}) \quad (6.24)$$

$$= -\log \sigma((2y - 1)z) \quad (6.25)$$

$$= \zeta((1 - 2y)z). \quad (6.26)$$

This derivation makes use of some properties from section 3.10. By rewriting the loss in terms of the softplus function, we can see that it saturates only when $(1 - 2y)z$ is very negative. Saturation thus occurs only when the model already has the right answer—when $y = 1$ and z is very positive, or $y = 0$ and z is very negative. When z has the wrong sign, the argument to the softplus function, $(1 - 2y)z$, may be simplified to $|z|$. As $|z|$ becomes large while z has the wrong sign, the softplus function asymptotes toward simply returning its argument $|z|$. The derivative with respect to z asymptotes to $\text{sign}(z)$, so, in the limit of extremely incorrect z , the softplus function does not shrink the gradient at all. This property is useful because it means that gradient-based learning can act to quickly correct a mistaken z .

When we use other loss functions, such as mean squared error, the loss can saturate anytime $\sigma(z)$ saturates. The sigmoid activation function saturates to 0 when z becomes very negative and saturates to 1 when z becomes very positive. The gradient can shrink too small to be useful for learning when this happens, whether the model has the correct answer or the incorrect answer. For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units.

Analytically, the logarithm of the sigmoid is always defined and finite, because the sigmoid returns values restricted to the open interval $(0, 1)$, rather than using the entire closed interval of valid probabilities $[0, 1]$. In software implementations, to avoid numerical problems, it is best to write the negative log-likelihood as a function of z , rather than as a function of $\hat{y} = \sigma(z)$. If the sigmoid function underflows to zero, then taking the logarithm of \hat{y} yields negative infinity.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function, which was used to represent a probability distribution over a binary variable.

Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions

can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

In the case of binary variables, we wished to produce a single number

$$\hat{y} = P(y = 1 \mid \mathbf{x}). \quad (6.27)$$

Because this number needed to lie between 0 and 1, and because we wanted the logarithm of the number to be well behaved for gradient-based optimization of the log-likelihood, we chose to instead predict a number $z = \log \tilde{P}(y = 1 \mid \mathbf{x})$. Exponentiating and normalizing gave us a Bernoulli distribution controlled by the sigmoid function.

To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\mathbf{y}}$, with $\hat{y}_i = P(y = i \mid \mathbf{x})$. We require not only that each element of \hat{y}_i be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the multinoulli distribution. First, a linear layer predicts unnormalized log probabilities:

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}, \quad (6.28)$$

where $z_i = \log \tilde{P}(y = i \mid \mathbf{x})$. The softmax function can then exponentiate and normalize \mathbf{z} to obtain the desired $\hat{\mathbf{y}}$. Formally, the softmax function is given by

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \quad (6.29)$$

As with the logistic sigmoid, the use of the `exp` function works well when training the softmax to output a target value y using maximum log-likelihood. In this case, we wish to maximize $\log P(y = i; \mathbf{z}) = \log \text{softmax}(\mathbf{z})_i$. Defining the softmax in terms of `exp` is natural because the `log` in the log-likelihood can undo the `exp` of the softmax:

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j). \quad (6.30)$$

The first term of equation 6.30 shows that the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of equation 6.30 becomes very small. When maximizing the log-likelihood, the first term encourages z_i to be pushed up, while the second term encourages all of \mathbf{z} to be pushed down. To gain some intuition for the second term, $\log \sum_j \exp(z_j)$, observe

that this term can be roughly approximated by $\max_j z_j$. This approximation is based on the idea that $\exp(z_k)$ is insignificant for any z_k that is noticeably less than $\max_j z_j$. The intuition we can gain from this approximation is that the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest input to the softmax, then the $-z_i$ term and the $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ terms will roughly cancel. This example will then contribute little to the overall training cost, which will be dominated by other examples that are not yet correctly classified.

So far we have discussed only a single example. Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set:

$$\text{softmax}(\mathbf{z}(\mathbf{x}; \boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^m \mathbf{1}_{y^{(j)}=i, \mathbf{x}^{(j)}=\mathbf{x}}}{\sum_{j=1}^m \mathbf{1}_{\mathbf{x}^{(j)}=\mathbf{x}}}. \quad (6.31)$$

Because maximum likelihood is a consistent estimator, this is guaranteed to happen as long as the model family is capable of representing the training distribution. In practice, limited model capacity and imperfect optimization will mean that the model is only able to approximate these fractions.

Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions (Bridle, 1990). To understand why these other loss functions can fail, we need to examine the softmax function itself.

Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. The softmax has multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

To see that the softmax function responds to the difference between its inputs, observe that the softmax output is invariant to adding the same scalar to all its inputs:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c). \quad (6.32)$$

Using this property, we can derive a numerically stable variant of the softmax:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_i z_i). \quad (6.33)$$

The reformulated version enables us to evaluate softmax with only small numerical errors, even when z contains extremely large or extremely negative numbers. Examining the numerically stable variant, we see that the softmax function is driven by the amount that its arguments deviate from $\max_i z_i$.

An output $\text{softmax}(\mathbf{z})_i$ saturates to 1 when the corresponding input is maximal ($z_i = \max_i z_i$) and z_i is much greater than all the other inputs. The output $\text{softmax}(\mathbf{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater. This is a generalization of the way that sigmoid units saturate and can cause similar difficulties for learning if the loss function is not designed to compensate for it.

The argument \mathbf{z} to the softmax function can be produced in two different ways. The most common is simply to have an earlier layer of the neural network output every element of \mathbf{z} , as described above using the linear layer $\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$. While straightforward, this approach actually overparametrizes the distribution. The constraint that the n outputs must sum to 1 means that only $n - 1$ parameters are necessary; the probability of the n -th value may be obtained by subtracting the first $n - 1$ probabilities from 1. We can thus impose a requirement that one element of \mathbf{z} be fixed. For example, we can require that $z_n = 0$. Indeed, this is exactly what the sigmoid unit does. Defining $P(y = 1 | \mathbf{x}) = \sigma(z)$ is equivalent to defining $P(y = 1 | \mathbf{x}) = \text{softmax}(\mathbf{z})_1$ with a two-dimensional \mathbf{z} and $z_1 = 0$. Both the $n - 1$ argument and the n argument approaches to the softmax can describe the same set of probability distributions but have different learning dynamics. In practice, there is rarely much difference between using the overparametrized version or the restricted version, and it is simpler to implement the overparametrized version.

From a neuroscientific point of view, it is interesting to think of the softmax as a way to create a form of competition between the units that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex. At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of **winner-take-all** (one of the outputs is nearly 1, and the others are nearly 0).

The name “softmax” can be somewhat confusing. The function is more closely related to the arg max function than the max function. The term “soft” derives from the fact that the softmax function is continuous and differentiable. The arg max function, with its result represented as a one-hot vector, is not continuous or differentiable. The softmax function thus provides a “softened” version of the arg max. The corresponding soft version of the maximum function is $\text{softmax}(\mathbf{z})^\top \mathbf{z}$.

It would perhaps be better to call the softmax function “softargmax,” but the current name is an entrenched convention.

6.2.2.4 Other Output Types

The linear, sigmoid, and softmax output units described above are the most common. Neural networks can generalize to almost any kind of output layer that we wish. The principle of maximum likelihood provides a guide for how to design a good cost function for nearly any kind of output layer.

In general, if we define a conditional distribution $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$, the principle of maximum likelihood suggests we use $-\log p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ as our cost function.

In general, we can think of the neural network as representing a function $f(\mathbf{x}; \boldsymbol{\theta})$. The outputs of this function are not direct predictions of the value \mathbf{y} . Instead, $f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ provides the parameters for a distribution over y . Our loss function can then be interpreted as $-\log p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$.

For example, we may wish to learn the variance of a conditional Gaussian for \mathbf{y} , given \mathbf{x} . In the simple case, where the variance σ^2 is a constant, there is a closed form expression because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations \mathbf{y} and their expected value. A computationally more expensive approach that does not require writing special-case code is to simply include the variance as one of the properties of the distribution $p(\mathbf{y} | \mathbf{x})$ that is controlled by $\boldsymbol{\omega} = f(\mathbf{x}; \boldsymbol{\theta})$. The negative log-likelihood $-\log p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$ will then provide a cost function with the appropriate terms necessary to make our optimization procedure incrementally learn the variance. In the simple case where the standard deviation does not depend on the input, we can make a new parameter in the network that is copied directly into $\boldsymbol{\omega}$. This new parameter might be σ itself or could be a parameter v representing σ^2 or it could be a parameter β representing $\frac{1}{\sigma^2}$, depending on how we choose to parametrize the distribution. We may wish our model to predict a different amount of variance in \mathbf{y} for different values of \mathbf{x} . This is called a **heteroscedastic** model. In the heteroscedastic case, we simply make the specification of the variance be one of the values output by $f(\mathbf{x}; \boldsymbol{\theta})$. A typical way to do this is to formulate the Gaussian distribution using precision, rather than variance, as described in equation 3.22. In the multivariate case, it is most common to use a diagonal precision matrix

$$\text{diag}(\boldsymbol{\beta}). \quad (6.34)$$

This formulation works well with gradient descent because the formula for the log-likelihood of the Gaussian distribution parametrized by $\boldsymbol{\beta}$ involves only multiplication by β_i and addition of $\log \beta_i$. The gradient of multiplication, addition,

and logarithm operations is well behaved. By comparison, if we parametrized the output in terms of variance, we would need to use division. The division function becomes arbitrarily steep near zero. While large gradients can help learning, arbitrarily large gradients usually result in instability. If we parametrized the output in terms of standard deviation, the log-likelihood would still involve division as well as squaring. The gradient through the squaring operation can vanish near zero, making it difficult to learn parameters that are squared. Regardless of whether we use standard deviation, variance, or precision, we must ensure that the covariance matrix of the Gaussian is positive definite. Because the eigenvalues of the precision matrix are the reciprocals of the eigenvalues of the covariance matrix, this is equivalent to ensuring that the precision matrix is positive definite. If we use a diagonal matrix, or a scalar times the diagonal matrix, then the only condition we need to enforce on the output of the model is positivity. If we suppose that \mathbf{a} is the raw activation of the model used to determine the diagonal precision, we can use the softplus function to obtain a positive precision vector: $\boldsymbol{\beta} = \zeta(\mathbf{a})$. This same strategy applies equally if using variance or standard deviation rather than precision or if using a scalar times identity rather than diagonal matrix.

It is rare to learn a covariance or precision matrix with richer structure than diagonal. If the covariance is full and conditional, then a parametrization must be chosen that guarantees positive definiteness of the predicted covariance matrix. This can be achieved by writing $\boldsymbol{\Sigma}(\mathbf{x}) = \mathbf{B}(\mathbf{x})\mathbf{B}^\top(\mathbf{x})$, where \mathbf{B} is an unconstrained square matrix. One practical issue if the matrix is full rank is that computing the likelihood is expensive, with a $d \times d$ matrix requiring $O(d^3)$ computation for the determinant and inverse of $\boldsymbol{\Sigma}(\mathbf{x})$ (or equivalently, and more commonly done, its eigendecomposition or that of $\mathbf{B}(\mathbf{x})$).

We often want to perform multimodal regression, that is, to predict real values from a conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ that can have several different peaks in \mathbf{y} space for the same value of \mathbf{x} . In this case, a Gaussian mixture is a natural representation for the output (Jacobs *et al.*, 1991; Bishop, 1994). Neural networks with Gaussian mixtures as their output are often called **mixture density networks**. A Gaussian mixture output with n components is defined by the conditional probability distribution:

$$p(\mathbf{y} \mid \mathbf{x}) = \sum_{i=1}^n p(c=i \mid \mathbf{x}) \mathcal{N}(\mathbf{y}; \boldsymbol{\mu}^{(i)}(\mathbf{x}), \boldsymbol{\Sigma}^{(i)}(\mathbf{x})). \quad (6.35)$$

The neural network must have three outputs: a vector defining $p(c=i \mid \mathbf{x})$, a matrix providing $\boldsymbol{\mu}^{(i)}(\mathbf{x})$ for all i , and a tensor providing $\boldsymbol{\Sigma}^{(i)}(\mathbf{x})$ for all i . These outputs must satisfy different constraints:

1. Mixture components $p(c = i \mid \mathbf{x})$: these form a multinoulli distribution over the n different components associated with latent variable¹ c , and can typically be obtained by a softmax over an n -dimensional vector, to guarantee that these outputs are positive and sum to 1.
2. Means $\boldsymbol{\mu}^{(i)}(\mathbf{x})$: these indicate the center or mean associated with the i -th Gaussian component and are unconstrained (typically with no nonlinearity at all for these output units). If \mathbf{y} is a d -vector, then the network must output an $n \times d$ matrix containing all n of these d -dimensional vectors. Learning these means with maximum likelihood is slightly more complicated than learning the means of a distribution with only one output mode. We only want to update the mean for the component that actually produced the observation. In practice, we do not know which component produced each observation. The expression for the negative log-likelihood naturally weights each example's contribution to the loss for each component by the probability that the component produced the example.
3. Covariances $\boldsymbol{\Sigma}^{(i)}(\mathbf{x})$: these specify the covariance matrix for each component i . As when learning a single Gaussian component, we typically use a diagonal matrix to avoid needing to compute determinants. As with learning the means of the mixture, maximum likelihood is complicated by needing to assign partial responsibility for each point to each mixture component. Gradient descent will automatically follow the correct process if given the correct specification of the negative log-likelihood under the mixture model.

It has been reported that gradient-based optimization of conditional Gaussian mixtures (on the output of neural networks) can be unreliable, in part because one gets divisions (by the variance) which can be numerically unstable (when some variance gets to be small for a particular example, yielding very large gradients). One solution is to **clip gradients** (see section 10.11.1), while another is to scale the gradients heuristically ([Uria et al., 2014](#)).

Gaussian mixture outputs are particularly effective in generative models of speech ([Schuster, 1999](#)) and movements of physical objects ([Graves, 2013](#)). The mixture density strategy gives a way for the network to represent multiple output modes and to control the variance of its output, which is crucial for obtaining

¹We consider c to be latent because we do not observe it in the data: given input \mathbf{x} and target \mathbf{y} , it is not possible to know with certainty which Gaussian component was responsible for \mathbf{y} , but we can imagine that \mathbf{y} was generated by picking one of them, and we can make that unobserved choice a random variable.

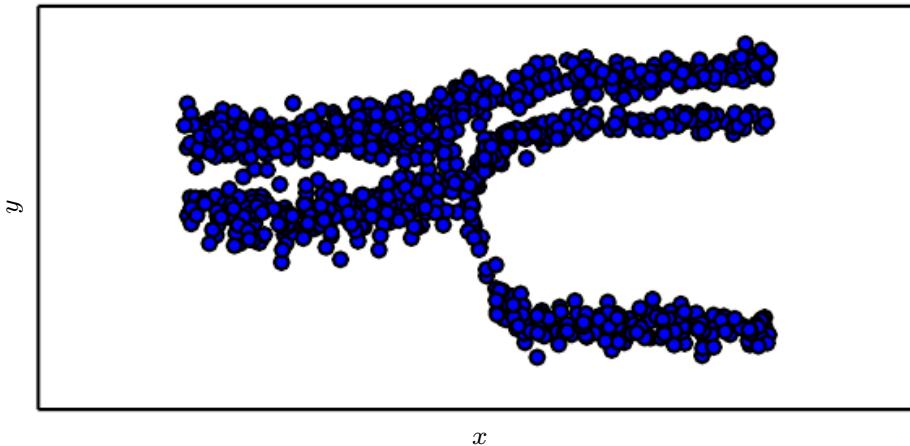


Figure 6.4: Samples drawn from a neural network with a mixture density output layer. The input x is sampled from a uniform distribution, and the output y is sampled from $p_{\text{model}}(y | x)$. The neural network is able to learn nonlinear mappings from the input to the parameters of the output distribution. These parameters include the probabilities governing which of three mixture components will generate the output as well as the parameters for each mixture component. Each mixture component is Gaussian with predicted mean and variance. All these aspects of the output distribution are able to vary with respect to the input x , and to do so in nonlinear ways.

a high degree of quality in these real-valued domains. An example of a mixture density network is shown in figure 6.4.

In general, we may wish to continue to model larger vectors \mathbf{y} containing more variables, and to impose richer and richer structures on these output variables. For example, if we want our neural network to output a sequence of characters that forms a sentence, we might continue to use the principle of maximum likelihood applied to our model $p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$. In this case, the model we use to describe \mathbf{y} would become complex enough to be beyond the scope of this chapter. In Chapter 10 we describe how to use recurrent neural networks to define such models over sequences, and in part III we describe advanced techniques for modeling arbitrary probability distributions.

6.3 Hidden Units

So far we have focused our discussion on design choices for neural networks that are common to most parametric machine learning models trained with gradient-based optimization. Now we turn to an issue that is unique to feedforward neural

networks: how to choose the type of hidden unit to use in the hidden layers of the model.

The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.

Rectified linear units are an excellent default choice of hidden unit. Many other types of hidden units are available. It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice). We describe here some of the basic intuitions motivating each type of hidden unit. These intuitions can help decide when to try out which unit. Predicting in advance which will work best is usually impossible. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Some of the hidden units included in this list are not actually differentiable at all input points. For example, the rectified linear function $g(z) = \max\{0, z\}$ is not differentiable at $z = 0$. This may seem like it invalidates g for use with a gradient-based learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks. This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly, as shown in figure 4.3. (These ideas are described further in chapter 8.) Because we do not expect training to actually reach a point where the gradient is $\mathbf{0}$, it is acceptable for the minima of the cost function to correspond to points with undefined gradient. Hidden units that are not differentiable are usually nondifferentiable at only a small number of points. In general, a function $g(z)$ has a left derivative defined by the slope of the function immediately to the left of z and a right derivative defined by the slope of the function immediately to the right of z . A function is differentiable at z only if both the left derivative and the right derivative are defined and equal to each other. The functions used in the context of neural networks usually have defined left derivatives and defined right derivatives. In the case of $g(z) = \max\{0, z\}$, the left derivative at $z = 0$ is 0, and the right derivative is 1. Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error. This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0. Instead, it was likely to be some small value ϵ that was rounded to 0. In some contexts, more theoretically pleasing justifications are available, but

these usually do not apply to neural network training. The important point is that in practice one can safely disregard the nondifferentiability of the hidden unit activation functions described below.

Unless indicated otherwise, most hidden units can be described as accepting a vector of inputs \mathbf{x} , computing an affine transformation $\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$, and then applying an element-wise nonlinear function $g(\mathbf{z})$. Most hidden units are distinguished from each other only by the choice of the form of the activation function $g(\mathbf{z})$.

6.3.1 Rectified Linear Units and Their Generalizations

Rectified linear units use the activation function $g(z) = \max\{0, z\}$.

These units are easy to optimize because they are so similar to linear units. The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active. The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.

Rectified linear units are typically used on top of an affine transformation:

$$\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}). \quad (6.36)$$

When initializing the parameters of the affine transformation, it can be a good practice to set all elements of \mathbf{b} to a small positive value, such as 0.1. Doing so makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

Several generalizations of rectified linear units exist. Most of these generalizations perform comparably to rectified linear units and occasionally perform better.

One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. Various generalizations of rectified linear units guarantee that they receive gradient everywhere.

Three generalizations of rectified linear units are based on using a nonzero slope α_i when $z_i < 0$: $h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$. **Absolute value**

rectification fixes $\alpha_i = -1$ to obtain $g(z) = |z|$. It is used for object recognition from images (Jarrett *et al.*, 2009), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination. Other generalizations of rectified linear units are more broadly applicable. A **leaky ReLU** (Maas *et al.*, 2013) fixes α_i to a small value like 0.01, while a **parametric ReLU**, or **PReLU**, treats α_i as a learnable parameter (He *et al.*, 2015).

Maxout units (Goodfellow *et al.*, 2013a) generalize rectified linear units further. Instead of applying an element-wise function $g(z)$, maxout units divide \mathbf{z} into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

$$g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j, \quad (6.37)$$

where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i , $\{(i-1)k+1, \dots, ik\}$. This provides a way of learning a piecewise linear function that responds to multiple directions in the input \mathbf{x} space.

A maxout unit can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as *learning the activation function* itself rather than just the relationship between units. With large enough k , a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, a maxout layer with two pieces can learn to implement the same function of the input \mathbf{x} as a traditional layer using the rectified linear activation function, the absolute value rectification function, or the leaky or parametric ReLU, or it can learn to implement a totally different function altogether. The maxout layer will of course be parametrized differently from any of these other layer types, so the learning dynamics will be different even in the cases where maxout learns to implement the same function of \mathbf{x} as one of the other layer types.

Each maxout unit is now parametrized by k weight vectors instead of just one, so maxout units typically need more regularization than rectified linear units. They can work well without regularization if the training set is large and the number of pieces per unit is kept low (Cai *et al.*, 2013).

Maxout units have a few other benefits. In some cases, one can gain some statistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.

Because each unit is driven by multiple filters, maxout units have some redundancy that helps them resist a phenomenon called **catastrophic forgetting**, in which neural networks forget how to perform tasks that they were trained on in

the past (Goodfellow *et al.*, 2014a).

Rectified linear units and all these generalizations of them are based on the principle that models are easier to optimize if their behavior is closer to linear. This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation—a particular straightforward kind of linear activation. This is discussed further in section 10.10.

6.3.2 Logistic Sigmoid and Hyperbolic Tangent

Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function

$$g(z) = \sigma(z) \quad (6.38)$$

or the hyperbolic tangent activation function

$$g(z) = \tanh(z). \quad (6.39)$$

These activation functions are closely related because $\tanh(z) = 2\sigma(2z) - 1$.

We have already seen sigmoid units as output units, used to predict the probability that a binary variable is 1. Unlike piecewise linear units, sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0. The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged. Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.

When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$. Because \tanh is similar to the identity function near 0, training a deep neural network $\hat{y} = \mathbf{w}^\top \tanh(\mathbf{U}^\top \tanh(\mathbf{V}^\top \mathbf{x}))$ resembles training a linear model $\hat{y} =$

$\mathbf{w}^\top \mathbf{U}^\top \mathbf{V}^\top \mathbf{x}$ as long as the activations of the network can be kept small. This makes training the tanh network easier.

Sigmoidal activation functions are more common in settings other than feed-forward networks. Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation.

6.3.3 Other Hidden Units

Many other types of hidden units are possible but are used less frequently.

In general, a wide variety of differentiable functions perform perfectly well. Many unpublished activation functions perform just as well as the popular ones. To provide a concrete example, we tested a feedforward network using $\mathbf{h} = \cos(\mathbf{W}\mathbf{x} + \mathbf{b})$ on the MNIST dataset and obtained an error rate of less than 1 percent, which is competitive with results obtained using more conventional activation functions. During research and development of new techniques, it is common to test many different activation functions and find that several variations on standard practice perform comparably. This means that usually new hidden unit types are published only if they are clearly demonstrated to provide a significant improvement. New hidden unit types that perform roughly comparably to known types are so common as to be uninteresting.

It would be impractical to list all the hidden unit types that have appeared in the literature. We highlight a few especially useful and distinctive ones.

One possibility is to not have an activation $g(z)$ at all. One can also think of this as using the identity function as the activation function. We have already seen that a linear unit can be useful as the output of a neural network. It may also be used as a hidden unit. If every layer of the neural network consists of only linear transformations, then the network as a whole will be linear. However, it is acceptable for some layers of the neural network to be purely linear. Consider a neural network layer with n inputs and p outputs, $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$. We may replace this with two layers, with one layer using weight matrix \mathbf{U} and the other using weight matrix \mathbf{V} . If the first layer has no activation function, then we have essentially factored the weight matrix of the original layer based on \mathbf{W} . The factored approach is to compute $\mathbf{h} = g(\mathbf{V}^\top \mathbf{U}^\top \mathbf{x} + \mathbf{b})$. If \mathbf{U} produces q outputs, then \mathbf{U} and \mathbf{V} together contain only $(n + p)q$ parameters, while \mathbf{W} contains np parameters. For small q , this can be a considerable saving in parameters. It comes at the cost of constraining the linear transformation to be low rank, but

these low-rank relationships are often sufficient. Linear hidden units thus offer an effective way of reducing the number of parameters in a network.

Softmax units are another kind of unit that is usually used as an output (as described in section 6.2.2.3) but may sometimes be used as a hidden unit. Softmax units naturally represent a probability distribution over a discrete variable with k possible values, so they may be used as a kind of switch. These kinds of hidden units are usually only used in more advanced architectures that explicitly learn to manipulate memory, as described in section 10.12.

A few other reasonably common hidden unit types include

- **Radial basis function** (RBF), unit: $h_i = \exp\left(-\frac{1}{\sigma_i^2} \|\mathbf{W}_{:,i} - \mathbf{x}\|^2\right)$. This function becomes more active as \mathbf{x} approaches a template $\mathbf{W}_{:,i}$. Because it saturates to 0 for most \mathbf{x} , it can be difficult to optimize.
- **Softplus**: $g(a) = \zeta(a) = \log(1+e^a)$. This is a smooth version of the rectifier, introduced by [Dugas et al. \(2001\)](#) for function approximation and by [Nair and Hinton \(2010\)](#) for the conditional distributions of undirected probabilistic models. [Glorot et al. \(2011a\)](#) compared the softplus and rectifier and found better results with the latter. The use of the softplus is generally discouraged. The softplus demonstrates that the performance of hidden unit types can be very counterintuitive—one might expect it to have an advantage over the rectifier due to being differentiable everywhere or due to saturating less completely, but empirically it does not.
- **Hard tanh**. This is shaped similarly to the tanh and the rectifier, but unlike the latter, it is bounded, $g(a) = \max(-1, \min(1, a))$. It was introduced by [Collobert \(2004\)](#).

Hidden unit design remains an active area of research, and many useful hidden unit types remain to be discovered.

6.4 Architecture Design

Another key design consideration for neural networks is determining the architecture. The word **architecture** refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.

Most neural networks are organized into groups of units called layers. Most neural network architectures arrange these layers in a chain structure, with each

layer being a function of the layer that preceded it. In this structure, the first layer is given by

$$\mathbf{h}^{(1)} = g^{(1)} \left(\mathbf{W}^{(1)\top} \mathbf{x} + \mathbf{b}^{(1)} \right); \quad (6.40)$$

the second layer is given by

$$\mathbf{h}^{(2)} = g^{(2)} \left(\mathbf{W}^{(2)\top} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} \right); \quad (6.41)$$

and so on.

In these chain-based architectures, the main architectural considerations are choosing the depth of the network and the width of each layer. As we will see, a network with even one hidden layer is sufficient to fit the training set. Deeper networks are often able to use far fewer units per layer and far fewer parameters, as well as frequently generalizing to the test set, but they also tend to be harder to optimize. The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.

6.4.1 Universal Approximation Properties and Depth

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models. Unfortunately, we often want our systems to learn nonlinear functions.

At first glance, we might presume that learning a nonlinear function requires designing a specialized model family for the kind of nonlinearity we want to learn. Fortunately, feedforward networks with hidden layers provide a universal approximation framework. Specifically, the **universal approximation theorem** (Hornik *et al.*, 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et al.*, 1990). The concept of Borel measurability is beyond the scope of this book; for our purposes it suffices to say that any continuous function on a closed and bounded subset of \mathbb{R}^n is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space

to another. While the original theorems were first stated in terms of units with activation functions that saturate for both very negative and very positive arguments, universal approximation theorems have also been proved for a wider class of activation functions, which includes the now commonly used rectified linear unit (Leshno *et al.*, 1993).

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function. We are not guaranteed, however, that the training algorithm will be able to *learn* that function. Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function as a result of overfitting. Recall from section 5.2.1 that the no free lunch theorem shows that there is no universally superior machine learning algorithm. Feedforward networks provide a universal system for representing functions in the sense that, given a function, there exists a feedforward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

According to the universal approximation theorem, there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. Barron (1993) provides some bounds on the size of a single-layer network needed to approximate a broad class of functions. Unfortunately, in the worst case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $\mathbf{v} \in \{0, 1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

Various families of functions can be approximated efficiently by an architecture with depth greater than some value d , but they require a much larger model if depth is restricted to be less than or equal to d . In many cases, the number of hidden units required by the shallow model is exponential in n . Such results

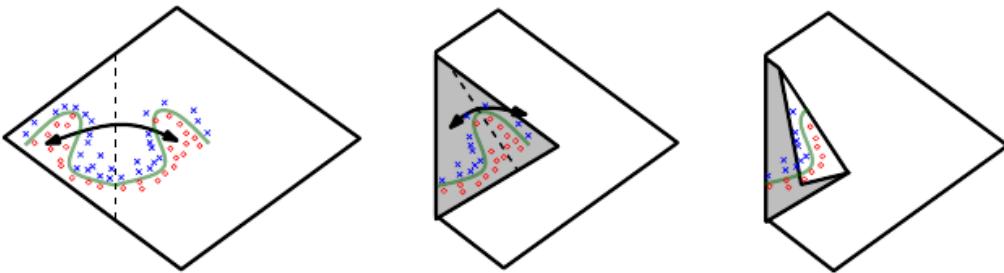


Figure 6.5: An intuitive, geometric explanation of the exponential advantage of deeper rectifier networks formally by Montufar *et al.* (2014). (*Left*) An absolute value rectification unit has the same output for every pair of mirror points in its input. The mirror axis of symmetry is given by the hyperplane defined by the weights and bias of the unit. A function computed on top of that unit (the green decision surface) will be a mirror image of a simpler pattern across that axis of symmetry. (*Center*) The function can be obtained by folding the space around the axis of symmetry. (*Right*) Another repeating pattern can be folded on top of the first (by another downstream unit) to obtain another symmetry (which is now repeated four times, with two hidden layers). Figure reproduced with permission from Montufar *et al.* (2014).

were first proved for models that do not resemble the continuous, differentiable neural networks used for machine learning but have since been extended to these models. The first results were for circuits of logic gates (Håstad, 1986). Later work extended these results to linear threshold units with nonnegative weights (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), and then to networks with continuous-valued activations (Maass, 1992; Maass *et al.*, 1994). Many modern neural networks use rectified linear units. Leshno *et al.* (1993) demonstrated that shallow networks with a broad family of non-polynomial activation functions, including rectified linear units, have universal approximation properties, but these results do not address the questions of depth or efficiency—they specify only that a sufficiently wide rectifier network could represent any function. Montufar *et al.* (2014) showed that functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network. More precisely, they showed that piecewise linear networks (which can be obtained from rectifier nonlinearities or maxout units) can represent functions with a number of regions that is exponential in the depth of the network. Figure 6.5 illustrates how a network with absolute value rectification creates mirror images of the function computed on top of some hidden unit, with respect to the input of that hidden unit. Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value nonlinearity). By composing these folding operations, we obtain an exponentially large number of piecewise linear regions that can capture all kinds of regular (e.g., repeating) patterns.

The main theorem in [Montufar *et al.* \(2014\)](#) states that the number of linear regions carved out by a deep rectifier network with d inputs, depth l , and n units per hidden layer is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right), \quad (6.42)$$

that is, exponential in depth l . In the case of maxout networks with k filters per unit, the number of linear regions is

$$O\left(k^{(l-1)+d}\right). \quad (6.43)$$

Of course, there is no guarantee that the kinds of functions we want to learn in applications of machine learning (and in particular for AI) share such a property.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks ([Bengio *et al.*, 2007](#); [Erhan *et al.*, 2009](#); [Bengio, 2009](#); [Mesnil *et al.*, 2011](#); [Ciresan *et al.*, 2012](#); [Krizhevsky *et al.*, 2012](#); [Sermanet *et al.*, 2013](#); [Farabet *et al.*, 2013](#); [Couprie *et al.*, 2013](#); [Kahou *et al.*, 2013](#); [Goodfellow *et al.*, 2014d](#); [Szegedy *et al.*, 2014a](#)). See figure 6.6 and figure 6.7 for examples of some of these empirical results. These results suggest that using deep architectures does indeed express a useful prior over the space of functions the model learns.

6.4.2 Other Architectural Considerations

So far we have described neural networks as being simple chains of layers, with the main considerations being the depth of the network and the width of each layer. In practice, neural networks show considerably more diversity.

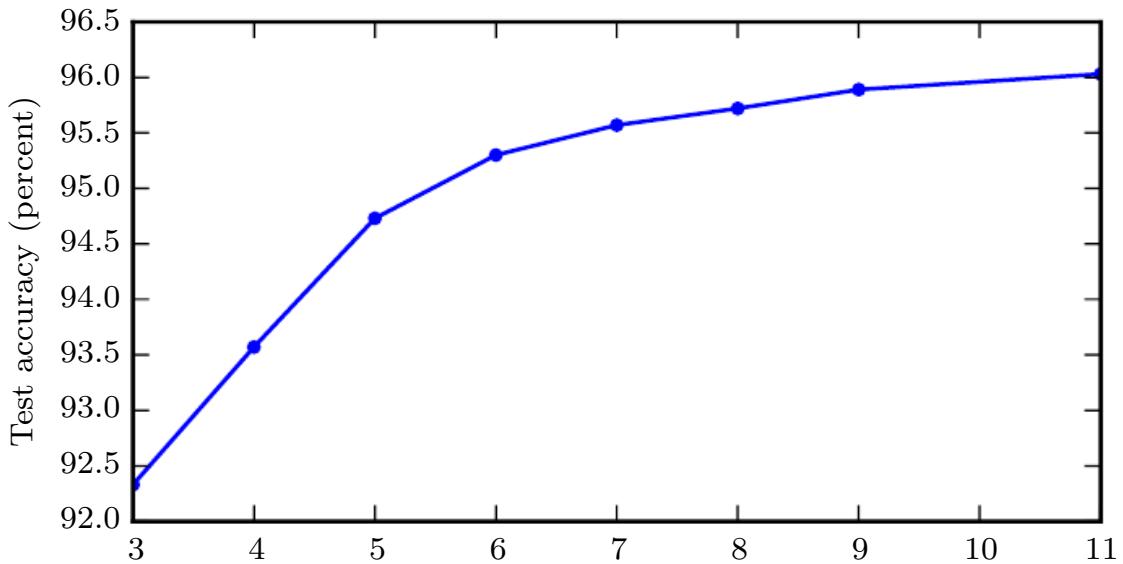


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from [Goodfellow et al. \(2014d\)](#). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

Many neural network architectures have been developed for specific tasks. Specialized architectures for computer vision called convolutional networks are described in chapter 9. Feedforward networks may also be generalized to the recurrent neural networks for sequence processing, described in chapter 10, which have their own architectural considerations.

In general, the layers need not be connected in a chain, even though this is the most common practice. Many architectures build a main chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i + 2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.

Another key consideration of architecture design is exactly how to connect a pair of layers to each other. In the default neural network layer described by a linear transformation via a matrix \mathbf{W} , every input unit is connected to every output unit. Many specialized networks in the chapters ahead have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer. These strategies for decreasing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network but are often highly problem dependent. For example, convolutional

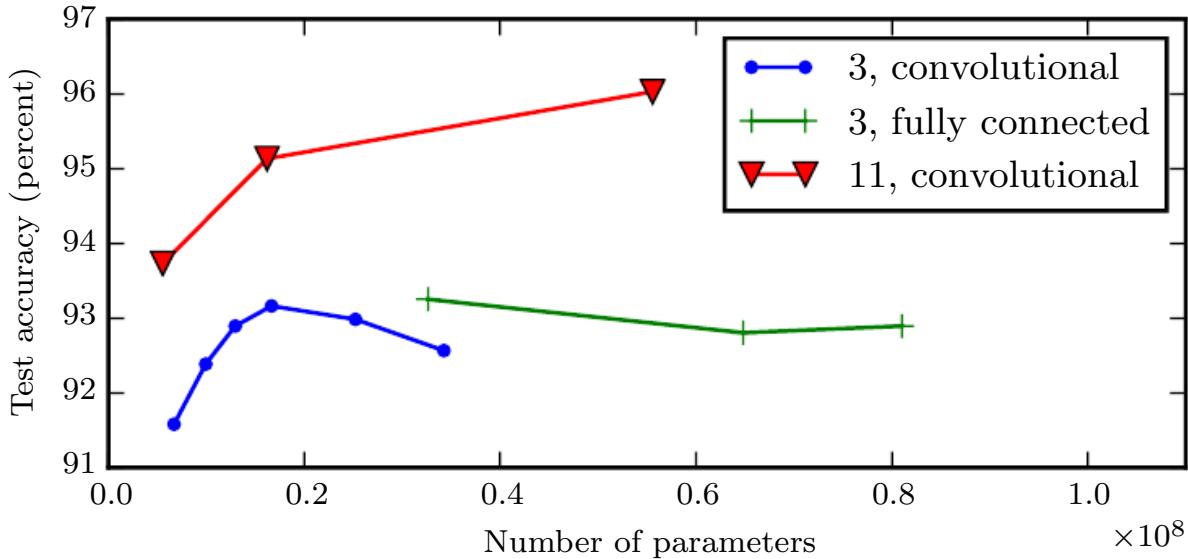


Figure 6.7: Effect of number of parameters. Deeper models tend to perform better. This is not merely because the model is larger. This experiment from [Goodfellow et al. \(2014d\)](#) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance, as illustrated in this figure. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

networks, described in chapter 9, use specialized patterns of sparse connections that are very effective for computer vision problems. In this chapter, it is difficult to give more specific advice concerning the architecture of a generic neural network. In subsequent chapters we develop the particular architectural strategies that have been found to work well for different application domains.

6.5 Back-Propagation and Other Differentiation Algorithms

When we use a feedforward neural network to accept an input \mathbf{x} and produce an output $\hat{\mathbf{y}}$, information flows forward through the network. The input \mathbf{x} provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\mathbf{y}}$. This is called **forward propagation**. During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The **back-propagation** algorithm (Rumelhart *et al.*, 1986a), often simply called **backprop**, allows the information from the cost to then flow backward through the network in order to compute the gradient.

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined). Specifically, we will describe how to compute the gradient $\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{y})$ for an arbitrary function f , where \mathbf{x} is a set of variables whose derivatives are desired, and \mathbf{y} is an additional set of variables that are inputs to the function but whose derivatives are not required. In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Many machine learning tasks involve computing other derivatives, either as part of the learning process, or to analyze the learned model. The back-propagation algorithm can be applied to these tasks as well and is not restricted to computing the gradient of the cost function with respect to the parameters. The idea of computing derivatives by propagating information through a network is very general and can be used to compute values such as the Jacobian of a function

f with multiple outputs. We restrict our description here to the most commonly used case, where f has a single output.

6.5.1 Computational Graphs

So far we have discussed neural networks with a relatively informal graph language. To describe the back-propagation algorithm more precisely, it is helpful to have a more precise **computational graph** language.

Many ways of formalizing computation as graphs are possible.

Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.

To formalize our graphs, we also need to introduce the idea of an **operation**. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together.

Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector. Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case in our description because it introduces many extra details that are not important to conceptual understanding.

If a variable y is computed by applying an operation to a variable x , then we draw a directed edge from x to y . We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Examples of computational graphs are shown in figure 6.8.

6.5.2 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then

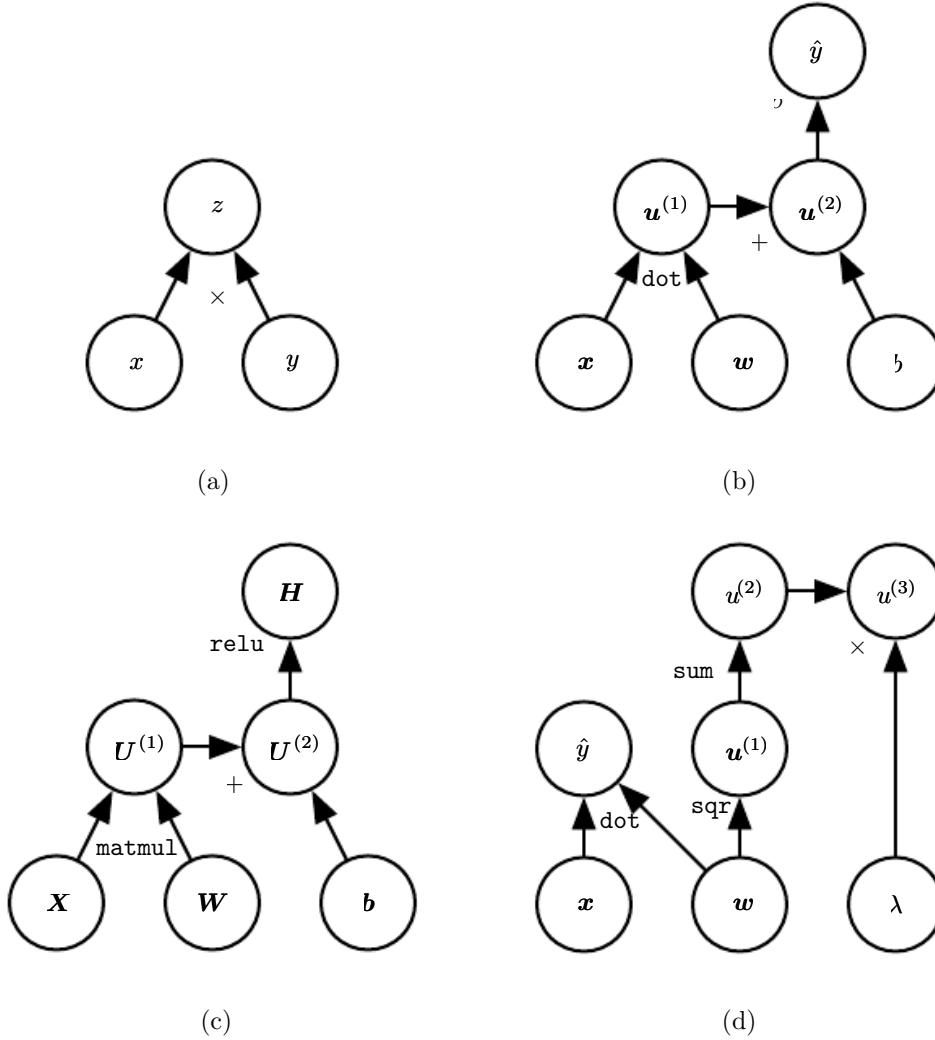


Figure 6.8: Examples of computational graphs. (a) The graph using the \times operation to compute $z = xy$. (b) The graph for the logistic regression prediction $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w} + b)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i -th such variable $\mathbf{u}^{(i)}$. (c) The computational graph for the expression $\mathbf{H} = \max\{0, \mathbf{X}\mathbf{W} + \mathbf{b}\}$, which computes a design matrix of rectified linear unit activations \mathbf{H} given a design matrix containing a minibatch of inputs \mathbf{X} . (d) Examples a–c applied at most one operation to each variable, but it is possible to apply more than one operation. Here we show a computation graph that applies more than one operation to the weights \mathbf{w} of a linear regression model. The weights are used to make both the prediction \hat{y} and the weight decay penalty $\lambda \sum_i w_i^2$.

the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (6.44)$$

We can generalize this beyond the scalar case. Suppose that $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g maps from \mathbb{R}^m to \mathbb{R}^n , and f maps from \mathbb{R}^n to \mathbb{R} . If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (6.45)$$

In vector notation, this may be equivalently written as

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^\top \nabla_{\mathbf{y}} z, \quad (6.46)$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g .

From this we see that the gradient of a variable \mathbf{x} can be obtained by multiplying a Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by a gradient $\nabla_{\mathbf{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Usually we apply the back-propagation algorithm to tensors of arbitrary dimensionality, not merely to vectors. Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor. We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

To denote the gradient of a value z with respect to a tensor \mathbf{X} , we write $\nabla_{\mathbf{X}} z$, just as if \mathbf{X} were a vector. The indices into \mathbf{X} now have multiple coordinates—for example, a 3-D tensor is indexed by three coordinates. We can abstract this away by using a single variable i to represent the complete tuple of indices. For all possible index tuples i , $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla_{\mathbf{x}} z)_i$ gives $\frac{\partial z}{\partial x_i}$. Using this notation, we can write the chain rule as it applies to tensors. If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \quad (6.47)$$

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. Actually evaluating that expression in a computer, however, introduces some extra considerations.

Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times. An example of how these repeated subexpressions arise is given in figure 6.9. In some cases, computing the same subexpression twice would simply be wasteful. For complicated graphs, there can be exponentially many of these wasted computations, making a naive implementation of the chain rule infeasible. In other cases, computing the same subexpression twice could be a valid way to reduce memory consumption at the cost of higher runtime.

We begin with a version of the back-propagation algorithm that specifies the actual gradient computation directly (algorithm 6.2 along with algorithm 6.1 for the associated forward computation), in the order it will actually be done and according to the recursive application of chain rule. One could either directly perform these computations or view the description of the algorithm as a symbolic specification of the computational graph for computing the back-propagation. However, this formulation does not make explicit the manipulation and the construction of the symbolic graph that performs the gradient computation. Such a formulation is presented in section 6.5.6, with algorithm 6.5, where we also generalize to nodes that contain arbitrary tensors.

First consider a computational graph describing how to compute a single scalar $u^{(n)}$ (say, the loss on a training example). This scalar is the quantity whose gradient we want to obtain, with respect to the n_i input nodes $u^{(1)}$ to $u^{(n_i)}$. In other words, we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1, 2, \dots, n_i\}$. In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model.

We will assume that the nodes of the graph have been ordered in such a way that we can compute their output one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$. As defined in algorithm 6.1, each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function

$$u^{(i)} = f(\mathbb{A}^{(i)}), \quad (6.48)$$

where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.

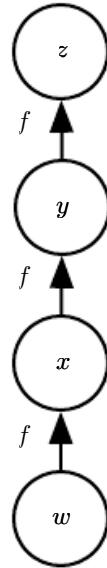


Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient. Let $w \in \mathbb{R}$ be the input to the graph. We use the same function $f : \mathbb{R} \rightarrow \mathbb{R}$ as the operation that we apply at every step of a chain: $x = f(w)$, $y = f(x)$, $z = f(y)$. To compute $\frac{\partial z}{\partial w}$, we apply equation 6.44 and obtain:

$$\frac{\partial z}{\partial w} \quad (6.49)$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \quad (6.50)$$

$$= f'(y) f'(x) f'(w) \quad (6.51)$$

$$= f'(f(f(w))) f'(f(w)) f'(w). \quad (6.52)$$

Equation 6.51 suggests an implementation in which we compute the value of $f(w)$ only once and store it in the variable x . This is the approach taken by the back-propagation algorithm. An alternative approach is suggested by equation 6.52, where the subexpression $f(w)$ appears more than once. In the alternative approach, $f(w)$ is recomputed each time it is needed. When the memory required to store the value of these expressions is low, the back-propagation approach of equation 6.51 is clearly preferable because of its reduced runtime. However, equation 6.52 is also a valid implementation of the chain rule and is useful when memory is limited.

That algorithm specifies the forward propagation computation, which we could put in a graph \mathcal{G} . To perform back-propagation, we can construct a computational graph that depends on \mathcal{G} and adds to it an extra set of nodes. These form a subgraph \mathcal{B} with one node per node of \mathcal{G} . Computation in \mathcal{B} proceeds in exactly the reverse of the order of computation in \mathcal{G} , and each node of \mathcal{B} computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$. This is done using the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \quad (6.53)$$

as specified by algorithm 6.2. The subgraph \mathcal{B} contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of \mathcal{G} . The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. In addition, a dot product is performed for each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$. To summarize, the amount of computation required for performing the back-propagation scales linearly with the number of edges in \mathcal{G} , where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition. Below, we generalize this analysis to tensor-valued nodes, which is just a way to group multiple scalar values in the same node and enable more

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector \mathbf{x} , and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

```
for  $i = 1, \dots, n_i$  do
     $u^{(i)} \leftarrow x_i$ 
end for
for  $i = n_i + 1, \dots, n$  do
     $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$ 
     $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$ 
end for
return  $u^{(n)}$ 
```

efficient implementations.

The back-propagation algorithm is designed to reduce the number of common subexpressions without regard to memory. Specifically, it performs on the order of one Jacobian product per node in the graph. This can be seen from the fact that backprop (algorithm 6.2) visits each edge from node $u^{(j)}$ to node $u^{(i)}$ of the graph exactly once in order to obtain the associated partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. Back-propagation thus avoids the exponential explosion in repeated subexpressions. Other algorithms may be able to avoid more subexpressions by performing simplifications on the computational graph, or may be able to conserve memory by recomputing rather than storing some subexpressions. We revisit these ideas after describing the back-propagation algorithm itself.

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \dots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`[$u^{(i)}$] will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`[$u^{(n)}$] $\leftarrow 1$

for $j = n - 1$ down to 1 **do**

The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

`grad_table`[$u^{(j)}$] $\leftarrow \sum_{i:j \in Pa(u^{(i)})} \text{grad_table}[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`[$u^{(i)}$] | $i = 1, \dots, n_i$ }

6.5.4 Back-Propagation Computation in Fully Connected MLP

To clarify the above definition of the back-propagation computation, let us consider the specific graph associated with a fully-connected multi layer MLP.

Algorithm 6.3 first shows the forward propagation, which maps parameters to the supervised loss $L(\hat{\mathbf{y}}, \mathbf{y})$ associated with a single (input,target) training example (\mathbf{x}, \mathbf{y}) , with $\hat{\mathbf{y}}$ the output of the neural network when \mathbf{x} is provided in input.

Algorithm 6.4 then shows the corresponding computation to be done for applying the back-propagation algorithm to this graph.

Algorithms 6.3 and 6.4 are demonstrations chosen to be simple and straightforward to understand. However, they are specialized to one specific problem.

Modern software implementations are based on the generalized form of back-propagation described in section 6.5.6 below, which can accommodate any computational graph by explicitly manipulating a data structure for representing symbolic computation.

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on the target \mathbf{y} (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J , the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters \mathbf{W} and \mathbf{b} . For simplicity, this demonstration uses only a single input example \mathbf{x} . Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l

Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model

Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$, the bias parameters of the model

Require: \mathbf{x} , the input to process

Require: \mathbf{y} , the target output

```
 $\mathbf{h}^{(0)} = \mathbf{x}$ 
for  $k = 1, \dots, l$  do
     $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$ 
     $\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$ 
end for
 $\hat{\mathbf{y}} = \mathbf{h}^{(l)}$ 
 $J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$ 
```

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input \mathbf{x} , a target \mathbf{y} . This computation yields the gradients on the activations $\mathbf{a}^{(k)}$ for each layer k , starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer’s output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

```
 $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$ 
for  $k = l, l - 1, \dots, 1$  do
```

Convert the gradient on the layer’s output into a gradient on the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

Compute gradients on weights and biases (including the regularization term, where needed):

$$\begin{aligned}\nabla_{\mathbf{b}^{(k)}} J &= \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta) \\ \nabla_{\mathbf{W}^{(k)}} J &= \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)\end{aligned}$$

Propagate the gradients w.r.t. the next lower-level hidden layer’s activations:

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top} \mathbf{g}$$

```
end for
```

6.5.5 Symbol-to-Symbol Derivatives

Algebraic expressions and computational graphs both operate on **symbols**, or variables that do not have specific values. These algebraic and graph-based representations are called **symbolic representations**. When we actually use or train a neural network, we must assign specific values to these symbols. We replace a symbolic input to the network \mathbf{x} with a specific **numeric** value, such as $[1.2, 3.765, -1.8]^\top$.

Some approaches to back-propagation take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values. We call this approach **symbol-to-number differentiation**. This is the approach used by libraries such as Torch (Collobert *et al.*, 2011b) and Caffe (Jia, 2013).

Another approach is to take a computational graph and add additional nodes

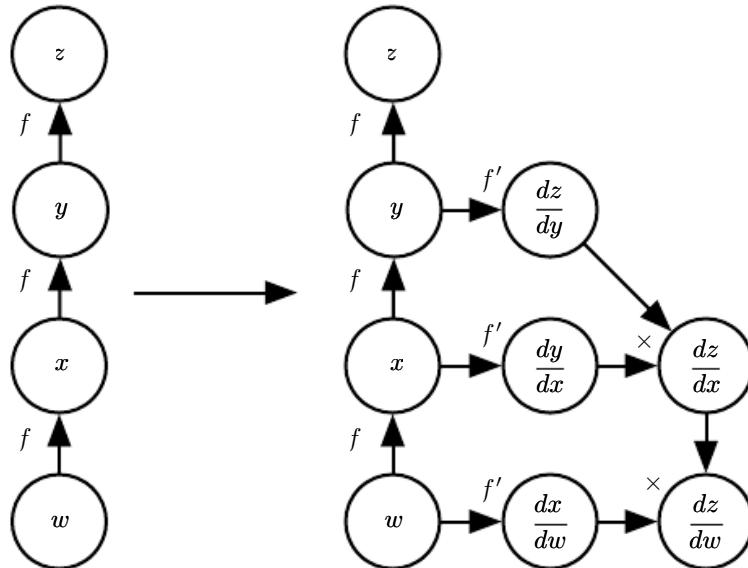


Figure 6.10: An example of the symbol-to-symbol approach to computing derivatives. In this approach, the back-propagation algorithm does not need to ever access any actual specific numeric values. Instead, it adds nodes to a computational graph describing how to compute these derivatives. A generic graph evaluation engine can later compute the derivatives for any specific numeric values. (*Left*) In this example, we begin with a graph representing $z = f(f(f(w)))$. (*Right*) We run the back-propagation algorithm, instructing it to construct the graph for the expression corresponding to $\frac{dz}{dw}$. In this example, we do not explain how the back-propagation algorithm works. The purpose is only to illustrate what the desired result is: a computational graph with a symbolic description of the derivative.

to the graph that provide a symbolic description of the desired derivatives. This is the approach taken by Theano ([Bergstra et al., 2010](#); [Bastien et al., 2012](#)) and TensorFlow ([Abadi et al., 2015](#)). An example of how it works is illustrated in figure 6.10. The primary advantage of this approach is that the derivatives are described in the same language as the original expression. Because the derivatives are just another computational graph, it is possible to run back-propagation again, differentiating the derivatives to obtain higher derivatives. (Computation of higher-order derivatives is described in section 6.5.10.)

We will use the latter approach and describe the back-propagation algorithm in terms of constructing a computational graph for the derivatives. Any subset of the graph may then be evaluated using specific numerical values at a later time. This allows us to avoid specifying exactly when each operation should be computed. Instead, a generic graph evaluation engine can evaluate every node as soon as its parents' values are available.

The description of the symbol-to-symbol based approach subsumes the symbol-

to-number approach. The symbol-to-number approach can be understood as performing exactly the same computations as are done in the graph built by the symbol-to-symbol approach. The key difference is that the symbol-to-number approach does not expose the graph.

6.5.6 General Back-Propagation

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors \mathbf{x} in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z . We continue multiplying by Jacobians, traveling backward through the graph in this way until we reach \mathbf{x} . For any node that may be reached by going backward from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

More formally, each node in the graph \mathcal{G} corresponds to a variable. To achieve maximum generality, we describe this variable as being a tensor \mathbf{V} . Tensors in general can have any number of dimensions. They subsume scalars, vectors, and matrices.

We assume that each variable \mathbf{V} is associated with the following subroutines:

- **get_operation(\mathbf{V})**: This returns the operation that computes \mathbf{V} , represented by the edges coming into \mathbf{V} in the computational graph. For example, there may be a Python or C++ class representing the matrix multiplication operation, and the `get_operation` function. Suppose we have a variable that is created by matrix multiplication, $\mathbf{C} = \mathbf{A}\mathbf{B}$. Then `get_operation(\mathbf{V})` returns a pointer to an instance of the corresponding C++ class.
- **get_consumers(\mathbf{V}, \mathcal{G})**: This returns the list of variables that are children of \mathbf{V} in the computational graph \mathcal{G} .
- **get_inputs(\mathbf{V}, \mathcal{G})**: This returns the list of variables that are parents of \mathbf{V} in the computational graph \mathcal{G} .

Each operation `op` is also associated with a `bprop` operation. This `bprop` operation can compute a Jacobian-vector product as described by equation 6.47. This is how the back-propagation algorithm is able to achieve great generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that it participates in. For example, we might use a matrix

multiplication operation to create a variable $\mathbf{C} = \mathbf{AB}$. Suppose that the gradient of a scalar z with respect to \mathbf{C} is given by \mathbf{G} . The matrix multiplication operation is responsible for defining two back-propagation rules, one for each of its input arguments. If we call the `bprop` method to request the gradient with respect to \mathbf{A} given that the gradient on the output is \mathbf{G} , then the `bprop` method of the matrix multiplication operation must state that the gradient with respect to \mathbf{A} is given by \mathbf{GB}^\top . Likewise, if we call the `bprop` method to request the gradient with respect to \mathbf{B} , then the matrix operation is responsible for implementing the `bprop` method and specifying that the desired gradient is given by $\mathbf{A}^\top\mathbf{G}$. The back-propagation algorithm itself does not need to know any differentiation rules. It only needs to call each operation's `bprop` rules with the right arguments. Formally, `op.bprop(inputs, X, G)` must return

$$\sum_i (\nabla_{\mathbf{X}} \text{op.f(inputs)}_i) G_i, \quad (6.54)$$

which is just an implementation of the chain rule as expressed in equation 6.47. Here, `inputs` is a list of inputs that are supplied to the operation, `op.f` is the mathematical function that the operation implements, \mathbf{X} is the input whose gradient we wish to compute, and \mathbf{G} is the gradient on the output of the operation.

The `op.bprop` method should always pretend that all its inputs are distinct from each other, even if they are not. For example, if the `mul` operator is passed two copies of x to compute x^2 , the `op.bprop` method should still return x as the derivative with respect to both inputs. The back-propagation algorithm will later add both of these arguments together to obtain $2x$, which is the correct total derivative on x .

Software implementations of back-propagation usually provide both the operations and their `bprop` methods, so that users of deep learning software libraries are able to back-propagate through graphs built using common operations like matrix multiplication, exponents, logarithms, and so on. Software engineers who build a new implementation of back-propagation or advanced users who need to add their own operation to an existing library must usually derive the `op.bprop` method for any new operations manually.

The back-propagation algorithm is formally described in algorithm 6.5.

In section 6.5.2, we explained that back-propagation was developed in order to avoid computing the same subexpression in the chain rule multiple times. The naive algorithm could have exponential runtime due to these repeated subexpressions. Now that we have specified the back-propagation algorithm, we can understand its computational cost. If we assume that each operation evaluation has roughly the

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

Require: \mathbb{T} , the target set of variables whose gradients must be computed.

Require: \mathcal{G} , the computational graph

Require: z , the variable to be differentiated

Let \mathcal{G}' be \mathcal{G} pruned to contain only nodes that are ancestors of z and descendants of nodes in \mathbb{T} .

Initialize `grad_table`, a data structure associating tensors to their gradients

$\text{grad_table}[z] \leftarrow 1$

for \mathbf{V} in \mathbb{T} **do**

`build_grad(V, G, G', grad_table)`

end for

Return `grad_table` restricted to \mathbb{T}

Algorithm 6.6 The inner loop subroutine `build_grad(V, G, G', grad_table)` of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: \mathbf{V} , the variable whose gradient should be added to \mathcal{G} and `grad_table`

Require: \mathcal{G} , the graph to modify

Require: \mathcal{G}' , the restriction of \mathcal{G} to nodes that participate in the gradient

Require: `grad_table`, a data structure mapping nodes to their gradients

if \mathbf{V} is in `grad_table` **then**

Return `grad_table[V]`

end if

$i \leftarrow 1$

for \mathbf{C} in `get_consumers(V, G')` **do**

`op` \leftarrow `get_operation(C)`

$\mathbf{D} \leftarrow \text{build_grad}(\mathbf{C}, \mathcal{G}, \mathcal{G}', \text{grad_table})$

$\mathbf{G}^{(i)} \leftarrow \text{op.bprop}(\text{get_inputs}(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D})$

$i \leftarrow i + 1$

end for

$\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$

`grad_table[V] = G`

Insert \mathbf{G} and the operations creating it into \mathcal{G}

Return \mathbf{G}

same cost, then we may analyze the computational cost in terms of the number of operations executed. Keep in mind here that we refer to an operation as the fundamental unit of our computational graph, which might actually consist of several arithmetic operations (for example, we might have a graph that treats matrix multiplication as a single operation). Computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations. Here we are counting operations in the computational graph, not individual operations executed by the underlying hardware, so it is important to remember that the runtime of each operation may be highly variable. For example, multiplying two matrices that each contain millions of entries might correspond to a single operation in the graph. We can see that computing the gradient requires at most $O(n^2)$ operations because the forward propagation stage will at worst execute all n nodes in the original graph (depending on which values we want to compute, we may not need to execute the entire graph). The back-propagation algorithm adds one Jacobian-vector product, which should be expressed with $O(1)$ nodes, per edge in the original graph. Because the computational graph is a directed acyclic graph it has at most $O(n^2)$ edges. For the kinds of graphs that are commonly used in practice, the situation is even better. Most neural network cost functions are roughly chain-structured, causing back-propagation to have $O(n)$ cost. This is far better than the naive approach, which might need to execute exponentially many nodes. This potentially exponential cost can be seen by expanding and rewriting the recursive chain rule (equation 6.53) nonrecursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path } (u^{(\pi_1)}, u^{(\pi_2)}, \dots, u^{(\pi_t)}), \\ \text{from } \pi_1=j \text{ to } \pi_t=n}} \prod_{k=2}^t \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \quad (6.55)$$

Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with the depth of the forward propagation graph. This large cost would be incurred because the same computation for $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ would be redone many times. To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order, back-propagation avoids repeating many common subexpressions. This table-filling strategy is sometimes called **dynamic programming**.

6.5.7 Example: Back-Propagation for MLP Training

As an example, we walk through the back-propagation algorithm as it is used to train a multilayer perceptron.

Here we develop a very simple multilayer perceptron with a single hidden layer. To train this model, we will use minibatch stochastic gradient descent. The back-propagation algorithm is used to compute the gradient of the cost on a single minibatch. Specifically, we use a minibatch of examples from the training set formatted as a design matrix \mathbf{X} and a vector of associated class labels \mathbf{y} . The network computes a layer of hidden features $\mathbf{H} = \max\{0, \mathbf{X}\mathbf{W}^{(1)}\}$. To simplify the presentation we do not use biases in this model. We assume that our graph language includes a `relu` operation that can compute $\max\{0, \mathbf{Z}\}$ element-wise. The predictions of the unnormalized log probabilities over classes are then given by $\mathbf{HW}^{(2)}$. We assume that our graph language includes a `cross_entropy` operation that computes the cross-entropy between the targets \mathbf{y} and the probability distribution defined by these unnormalized log probabilities. The resulting cross-entropy defines the cost J_{MLE} . Minimizing this cross-entropy performs maximum likelihood estimation of the classifier. However, to make this example more realistic, we also include a regularization term. The total cost

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \quad (6.56)$$

consists of the cross-entropy and a weight decay term with coefficient λ . The computational graph is illustrated in figure 6.11.

The computational graph for the gradient of this example is large enough that it would be tedious to draw or to read. This demonstrates one of the benefits of the back-propagation algorithm, which is that it can automatically generate gradients that would be straightforward but tedious for a software engineer to derive manually.

We can roughly trace out the behavior of the back-propagation algorithm by looking at the forward propagation graph in figure 6.11. To train, we wish to compute both $\nabla_{\mathbf{W}^{(1)}} J$ and $\nabla_{\mathbf{W}^{(2)}} J$. There are two different paths leading backward from J to the weights: one through the cross-entropy cost, and one through the weight decay cost. The weight decay cost is relatively simple; it will always contribute $2\lambda\mathbf{W}^{(i)}$ to the gradient on $\mathbf{W}^{(i)}$.

The other path through the cross-entropy cost is slightly more complicated. Let \mathbf{G} be the gradient on the unnormalized log probabilities $\mathbf{U}^{(2)}$ provided by the `cross_entropy` operation. The back-propagation algorithm now needs to

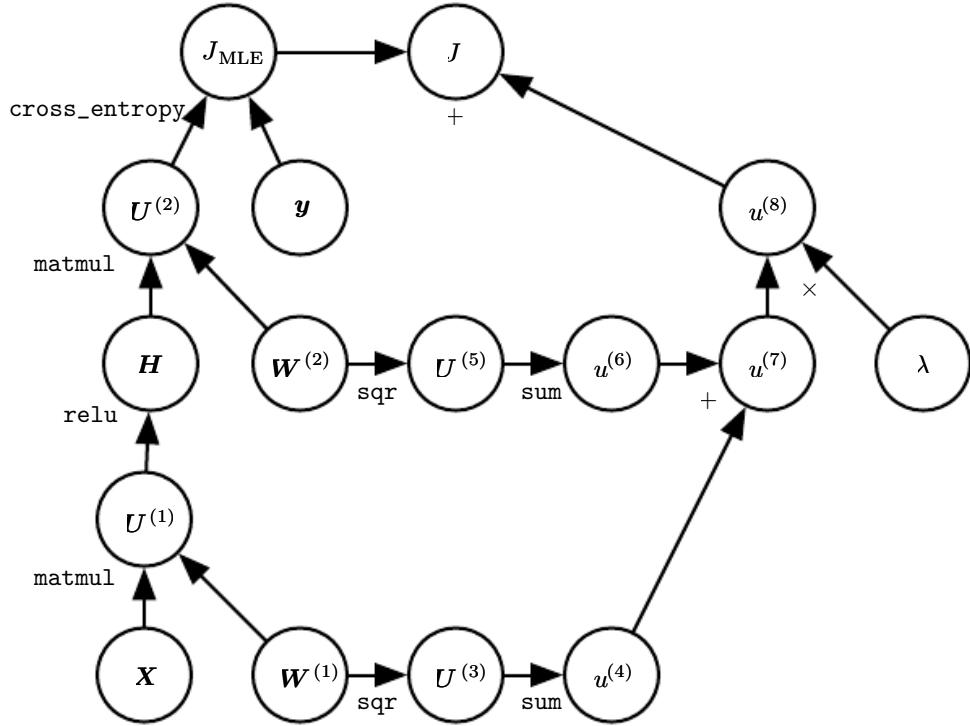


Figure 6.11: The computational graph used to compute the cost to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

explore two different branches. On the shorter branch, it adds $\mathbf{H}^\top \mathbf{G}$ to the gradient on $\mathbf{W}^{(2)}$, using the back-propagation rule for the second argument to the matrix multiplication operation. The other branch corresponds to the longer chain descending further along the network. First, the back-propagation algorithm computes $\nabla_{\mathbf{H}} J = \mathbf{G} \mathbf{W}^{(2)\top}$ using the back-propagation rule for the first argument to the matrix multiplication operation. Next, the `relu` operation uses its back-propagation rule to zero out components of the gradient corresponding to entries of $\mathbf{U}^{(1)}$ that are less than 0. Let the result be called \mathbf{G}' . The last step of the back-propagation algorithm is to use the back-propagation rule for the second argument of the `matmul` operation to add $\mathbf{X}^\top \mathbf{G}'$ to the gradient on $\mathbf{W}^{(1)}$.

After these gradients have been computed, the gradient descent algorithm, or another optimization algorithm, uses these gradients to update the parameters.

For the MLP, the computational cost is dominated by the cost of matrix multiplication. During the forward propagation stage, we multiply by each weight matrix, resulting in $O(w)$ multiply-adds, where w is the number of weights. During the backward propagation stage, we multiply by the transpose of each weight matrix, which has the same computational cost. The main memory cost of the algorithm is that we need to store the input to the nonlinearity of the hidden layer.

This value is stored from the time it is computed until the backward pass has returned to the same point. The memory cost is thus $O(mn_h)$, where m is the number of examples in the minibatch and n_h is the number of hidden units.

6.5.8 Complications

Our description of the back-propagation algorithm here is simpler than the implementations actually used in practice.

As noted above, we have restricted the definition of an operation to be a function that returns a single tensor. Most software implementations need to support operations that can return more than one tensor. For example, if we wish to compute both the maximum value in a tensor and the index of that value, it is best to compute both in a single pass through memory, so it is most efficient to implement this procedure as a single operation with two outputs.

We have not described how to control the memory consumption of back-propagation. Back-propagation often involves summation of many tensors together. In the naive approach, each of these tensors would be computed separately, then all of them would be added in a second step. The naive approach has an overly high memory bottleneck that can be avoided by maintaining a single buffer and adding each value to that buffer as it is computed.

Real-world implementations of back-propagation also need to handle various data types, such as 32-bit floating point, 64-bit floating point, and integer values. The policy for handling each of these types takes special care to design.

Some operations have undefined gradients, and it is important to track these cases and determine whether the gradient requested by the user is undefined.

Various other technicalities make real-world differentiation more complicated. These technicalities are not insurmountable, and this chapter has described the key intellectual tools needed to compute derivatives, but it is important to be aware that many more subtleties exist.

6.5.9 Differentiation outside the Deep Learning Community

The deep learning community has been somewhat isolated from the broader computer science community and has largely developed its own cultural attitudes concerning how to perform differentiation. More generally, the field of **automatic differentiation** is concerned with how to compute derivatives algorithmically. The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called **reverse**

mode accumulation. Other approaches evaluate the subexpressions of the chain rule in different orders. In general, determining the order of evaluation that results in the lowest computational cost is a difficult problem. Finding the optimal sequence of operations to compute the gradient is NP-complete (Naumann, 2008), in the sense that it may require simplifying algebraic expressions into their least expensive form.

For example, suppose we have variables p_1, p_2, \dots, p_n representing probabilities, and variables z_1, z_2, \dots, z_n representing unnormalized log probabilities. Suppose we define

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \quad (6.57)$$

where we build the softmax function out of exponentiation, summation and division operations, and construct a cross-entropy loss $J = -\sum_i p_i \log q_i$. A human mathematician can observe that the derivative of J with respect to z_i takes a very simple form: $q_i - p_i$. The back-propagation algorithm is not capable of simplifying the gradient this way and will instead explicitly propagate gradients through all the logarithm and exponentiation operations in the original graph. Some software libraries such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) are able to perform some kinds of algebraic substitution to improve over the graph proposed by the pure back-propagation algorithm.

When the forward graph \mathcal{G} has a single output node and each partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ can be computed with a constant amount of computation, back-propagation guarantees that the number of computations for the gradient computation is of the same order as the number of computations for the forward computation: this can be seen in algorithm 6.2, because each local partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ needs to be computed only once along with an associated multiplication and addition for the recursive chain-rule formulation (equation 6.53). The overall computation is therefore $O(\# \text{ edges})$. It can potentially be reduced, however, by simplifying the computational graph constructed by back-propagation, and this is an NP-complete task. Implementations such as Theano and TensorFlow use heuristics based on matching known simplification patterns to iteratively attempt to simplify the graph. We defined back-propagation only for computing a scalar output gradient, but back-propagation can be extended to compute a Jacobian (either of k different scalar nodes in the graph, or of a tensor-valued node containing k values). A naive implementation may then need k times more computation: for each scalar internal node in the original forward graph, the naive implementation computes k gradients instead of a single gradient. When the number of outputs of the graph is larger than the number of inputs, it is sometimes preferable to use another form of automatic differentiation called **forward mode accumulation**. Forward mode

accumulation has been proposed for obtaining real-time computation of gradients in recurrent networks, for example (Williams and Zipser, 1989). This approach also avoids the need to store the values and gradients for the whole graph, trading off computational efficiency for memory. The relationship between forward mode and backward mode is analogous to the relationship between left-multiplying versus right-multiplying a sequence of matrices, such as

$$\mathbf{A}\mathbf{B}\mathbf{C}\mathbf{D}, \quad (6.58)$$

where the matrices can be thought of as Jacobian. For example, if \mathbf{D} is a column vector while \mathbf{A} has many rows, the graph will have a single output and many inputs, and starting the multiplications from the end and going backward requires only matrix-vector products. This order corresponds to the backward mode. Instead, starting to multiply from the left would involve a series of matrix-matrix products, which makes the whole computation much more expensive. If \mathbf{A} has fewer rows than \mathbf{D} has columns, however, it is cheaper to run the multiplications left-to-right, corresponding to the forward mode.

In many communities outside machine learning, it is more common to implement differentiation software that acts directly on traditional programming language code, such as Python or C code, and automatically generates programs that differentiate functions written in these languages. In the deep learning community, computational graphs are usually represented by explicit data structures created by specialized libraries. The specialized approach has the drawback of requiring the library developer to define the `bprop` methods for every operation and limiting the user of the library to only those operations that have been defined. Yet the specialized approach also has the benefit of allowing customized back-propagation rules to be developed for each operation, enabling the developer to improve speed or stability in nonobvious ways that an automatic procedure would presumably be unable to replicate.

Back-propagation is therefore not the only way or the optimal way of computing the gradient, but it is a practical method that continues to serve the deep learning community well. In the future, differentiation technology for deep networks may improve as deep learning practitioners become more aware of advances in the broader field of automatic differentiation.

6.5.10 Higher-Order Derivatives

Some software frameworks support the use of higher-order derivatives. Among the deep learning software frameworks, this includes at least Theano and TensorFlow.

These libraries use the same kind of data structure to describe the expressions for derivatives as they use to describe the original function being differentiated. This means that the symbolic differentiation machinery can be applied to derivatives.

In the context of deep learning, it is rare to compute a single second derivative of a scalar function. Instead, we are usually interested in properties of the Hessian matrix. If we have a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, then the Hessian matrix is of size $n \times n$. In typical deep learning applications, n will be the number of parameters in the model, which could easily number in the billions. The entire Hessian matrix is thus infeasible to even represent.

Instead of explicitly computing the Hessian, the typical deep learning approach is to use **Krylov methods**. Krylov methods are a set of iterative techniques for performing various operations, such as approximately inverting a matrix or finding approximations to its eigenvectors or eigenvalues, without using any operation other than matrix-vector products.

To use Krylov methods on the Hessian, we only need to be able to compute the product between the Hessian matrix \mathbf{H} and an arbitrary vector \mathbf{v} . A straightforward technique (Christianson, 1992) for doing so is to compute

$$\mathbf{H}\mathbf{v} = \nabla_{\mathbf{x}} \left[(\nabla_{\mathbf{x}} f(\mathbf{x}))^\top \mathbf{v} \right]. \quad (6.59)$$

Both gradient computations in this expression may be computed automatically by the appropriate software library. Note that the outer gradient expression takes the gradient of a function of the inner gradient expression.

If \mathbf{v} is itself a vector produced by a computational graph, it is important to specify that the automatic differentiation software should not differentiate through the graph that produced \mathbf{v} .

While computing the Hessian is usually not advisable, it is possible to do with Hessian vector products. One simply computes $\mathbf{H}\mathbf{e}^{(i)}$ for all $i = 1, \dots, n$, where $\mathbf{e}^{(i)}$ is the one-hot vector with $e_i^{(i)} = 1$ and all other entries are equal to 0.

6.6 Historical Notes

Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation. From this point of view, the modern feedforward network is the culmination of centuries of progress on the general function approximation task.

The chain rule that underlies the back-propagation algorithm was invented in the seventeenth century (Leibniz, 1676; L'Hôpital, 1696). Calculus and algebra

have long been used to solve optimization problems in closed form, but gradient descent was not introduced as a technique for iteratively approximating the solution to optimization problems until the nineteenth century ([Cauchy, 1847](#)).

Beginning in the 1940s, these function approximation techniques were used to motivate machine learning models such as the perceptron. However, the earliest models were based on linear models. Critics including Marvin Minsky pointed out several of the flaws of the linear model family, such as its inability to learn the XOR function, which led to a backlash against the entire neural network approach.

Learning nonlinear functions required the development of a multilayer perceptron and a means of computing the gradient through such a model. Efficient applications of the chain rule based on dynamic programming began to appear in the 1960s and 1970s, mostly for control applications ([Kelley, 1960](#); [Bryson and Denham, 1961](#); [Dreyfus, 1962](#); [Bryson and Ho, 1969](#); [Dreyfus, 1973](#)) but also for sensitivity analysis ([Linnainmaa, 1976](#)). [Werbos \(1981\)](#) proposed applying these techniques to training artificial neural networks. The idea was finally developed in practice after being independently rediscovered in different ways ([LeCun, 1985](#); [Parker, 1985](#); [Rumelhart *et al.*, 1986a](#)). The book **Parallel Distributed Processing** presented the results of some of the first successful experiments with back-propagation in a chapter ([Rumelhart *et al.*, 1986b](#)) that contributed greatly to the popularization of back-propagation and initiated a very active period of research in multilayer neural networks. The ideas put forward by the authors of that book, particularly by Rumelhart and Hinton, go much beyond back-propagation. They include crucial ideas about the possible computational implementation of several central aspects of cognition and learning, which came under the name “connectionism” because of the importance this school of thought places on the connections between neurons as the locus of learning and memory. In particular, these ideas include the notion of distributed representation ([Hinton *et al.*, 1986](#)).

Following the success of back-propagation, neural network research gained popularity and reached a peak in the early 1990s. Afterwards, other machine learning techniques became more popular until the modern deep learning renaissance that began in 2006.

The core ideas behind modern feedforward networks have not changed substantially since the 1980s. The same back-propagation algorithm and the same approaches to gradient descent are still in use. Most of the improvement in neural network performance from 1986 to 2015 can be attributed to two factors. First, larger datasets have reduced the degree to which statistical generalization is a challenge for neural networks. Second, neural networks have become much larger, because of more powerful computers and better software infrastructure. A small

number of algorithmic changes have also improved the performance of neural networks noticeably.

One of these algorithmic changes was the replacement of mean squared error with the cross-entropy family of loss functions. Mean squared error was popular in the 1980s and 1990s but was gradually replaced by cross-entropy losses and the principle of maximum likelihood as ideas spread between the statistics community and the machine learning community. The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using the mean squared error loss.

The other major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piecewise linear hidden units, such as rectified linear units. Rectification using the $\max\{0, z\}$ function was introduced in early neural network models and dates back at least as far as the cognitron and neocognitron (Fukushima, 1975, 1980). These early models did not use rectified linear units but instead applied rectification to nonlinear functions. Despite the early popularity of rectification, it was largely replaced by sigmoids in the 1980s, perhaps because sigmoids perform better when neural networks are very small. As of the early 2000s, rectified linear units were avoided because of a somewhat superstitious belief that activation functions with nondifferentiable points must be avoided. This began to change in about 2009. Jarrett *et al.* (2009) observed that “using a rectifying nonlinearity is the single most important factor in improving the performance of a recognition system,” among several different factors of neural network architecture design.

For small datasets, Jarrett *et al.* (2009) observed that using rectifying nonlinearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, enabling the classifier layer at the top to learn how to map different feature vectors to class identities.

When more data is available, learning begins to extract enough useful knowledge to exceed the performance of randomly chosen parameters. Glorot *et al.* (2011a) showed that learning is far easier in deep rectified linear networks than in deep networks that have curvature or two-sided saturation in their activation functions.

Rectified linear units are also of historical interest because they show that neuroscience has continued to have an influence on the development of deep learning algorithms. Glorot *et al.* (2011a) motivated rectified linear units from biological considerations. The half-rectifying nonlinearity was intended to capture these properties of biological neurons: (1) For some inputs, biological neurons

are completely inactive. (2) For some inputs, a biological neuron’s output is proportional to its input. (3) Most of the time, biological neurons operate in the regime where they are inactive (i.e., they should have **sparse activations**).

When the modern resurgence of deep learning began in 2006, feedforward networks continued to have a bad reputation. From about 2006 to 2012, it was widely believed that feedforward networks would not perform well unless they were assisted by other models, such as probabilistic models. Today, it is now known that with the right resources and engineering practices, feedforward networks perform very well. Today, gradient-based learning in feedforward networks is used as a tool to develop probabilistic models, such as the variational autoencoder and generative adversarial networks, described in chapter 20. Rather than being viewed as an unreliable technology that must be supported by other techniques, gradient-based learning in feedforward networks has been viewed since 2012 as a powerful technology that can be applied to many other machine learning tasks. In 2006, the community used unsupervised learning to support supervised learning, and now, ironically, it is more common to use supervised learning to support unsupervised learning.

Feedforward networks continue to have unfulfilled potential. In the future, we expect they will be applied to many more tasks, and that advances in optimization algorithms and model design will improve their performance even further. This chapter has primarily described the neural network family of models. In the subsequent chapters, we turn to how to use these models—how to regularize and train them.

Chapter 7

Regularization for Deep Learning

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization. A great many forms of regularization are available to the deep learning practitioner. In fact, developing more effective regularization strategies has been one of the major research efforts in the field.

Chapter 5 introduced the basic concepts of generalization, underfitting, overfitting, bias, variance and regularization. If you are not already familiar with these notions, please refer to that chapter before continuing with this one.

In this chapter, we describe regularization in more detail, focusing on regularization strategies for deep models or models that may be used as building blocks to form deep models.

Some sections of this chapter deal with standard concepts in machine learning. If you are already familiar with these concepts, feel free to skip the relevant sections. However, most of this chapter is concerned with the extension of these basic concepts to the particular case of neural networks.

In section 5.2.2, we defined regularization as “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.” There are many regularization strategies. Some put extra constraints on a machine learning model, such as adding restrictions on the parameter values. Some add extra terms in the objective function that can be thought of as corresponding to a soft constraint on the parameter values. If chosen carefully, these extra constraints and penalties can lead to improved performance on the

test set. Sometimes these constraints and penalties are designed to encode specific kinds of prior knowledge. Other times, these constraints and penalties are designed to express a generic preference for a simpler model class in order to promote generalization. Sometimes penalties and constraints are necessary to make an underdetermined problem determined. Other forms of regularization, known as ensemble methods, combine multiple hypotheses that explain the training data.

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias. When we discussed generalization and overfitting in chapter 5, we focused on three situations, where the model family being trained either (1) excluded the true data-generating process—corresponding to underfitting and inducing bias, or (2) matched the true data-generating process, or (3) included the generating process but also many other possible generating processes—the overfitting regime where variance rather than bias dominates the estimation error. The goal of regularization is to take a model from the third regime into the second regime.

In practice, an overly complex model family does not necessarily include the target function or the true data-generating process, or even a close approximation of either. We almost never have access to the true data-generating process so we can never know for sure if the model family being estimated includes the generating process or not. Most applications of deep learning algorithms, however, are to domains where the true data-generating process is almost certainly outside the model family. Deep learning algorithms are typically applied to extremely complicated domains such as images, audio sequences and text, for which the true generation process essentially involves simulating the entire universe. To some extent, we are always trying to fit a square peg (the data-generating process) into a round hole (our model family).

What this means is that controlling the complexity of the model is not a simple matter of finding the model of the right size, with the right number of parameters. Instead, we might find—and indeed in practical deep learning scenarios, we almost always do find—that the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately.

We now review several strategies for how to create such a large, deep regularized model.

7.1 Parameter Norm Penalties

Regularization has been used for decades prior to the advent of deep learning. Linear models such as linear regression and logistic regression allow simple, straightforward, and effective regularization strategies.

Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J . We denote the regularized objective function by \tilde{J} :

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}), \quad (7.1)$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω , relative to the standard objective function J . Setting α to 0 results in no regularization. Larger values of α correspond to more regularization.

When our training algorithm minimizes the regularized objective function \tilde{J} it will decrease both the original objective J on the training data and some measure of the size of the parameters $\boldsymbol{\theta}$ (or some subset of the parameters). Different choices for the parameter norm Ω can result in different solutions being preferred. In this section, we discuss the effects of the various norms when used as penalties on the model parameters.

Before delving into the regularization behavior of different norms, we note that for neural networks, we typically choose to use a parameter norm penalty Ω that penalizes *only the weights* of the affine transformation at each layer and leaves the biases unregularized. The biases typically require less data than the weights to fit accurately. Each weight specifies how two variables interact. Fitting the weight well requires observing both variables in a variety of conditions. Each bias controls only a single variable. This means that we do not induce too much variance by leaving the biases unregularized. Also, regularizing the bias parameters can introduce a significant amount of underfitting. We therefore use the vector \mathbf{w} to indicate all the weights that should be affected by a norm penalty, while the vector $\boldsymbol{\theta}$ denotes all the parameters, including both \mathbf{w} and the unregularized parameters.

In the context of neural networks, it is sometimes desirable to use a separate penalty with a different α coefficient for each layer of the network. Because it can be expensive to search for the correct value of multiple hyperparameters, it is still reasonable to use the same weight decay at all layers just to reduce the size of search space.

7.1.1 L^2 Parameter Regularization

We have already seen, in section 5.2.2, one of the simplest and most common kinds of parameter norm penalty: the L^2 parameter norm penalty commonly known as **weight decay**. This regularization strategy drives the weights closer to the origin¹ by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2} \|\boldsymbol{\theta}\|_2^2$ to the objective function. In other academic communities, L^2 regularization is also known as **ridge regression** or **Tikhonov regularization**.

We can gain some insight into the behavior of weight decay regularization by studying the gradient of the regularized objective function. To simplify the presentation, we assume no bias parameter, so $\boldsymbol{\theta}$ is just \mathbf{w} . Such a model has the following total objective function:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.2)$$

with the corresponding parameter gradient

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (7.3)$$

To take a single gradient step to update the weights, we perform this update:

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon (\alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})). \quad (7.4)$$

Written another way, the update is

$$\mathbf{w} \leftarrow (1 - \epsilon \alpha) \mathbf{w} - \epsilon \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (7.5)$$

We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update. This describes what happens in a single step. But what happens over the entire course of training?

We will further simplify the analysis by making a quadratic approximation to the objective function in the neighborhood of the value of the weights that obtains minimal unregularized training cost, $\mathbf{w}^* = \arg \min_{\mathbf{w}} J(\mathbf{w})$. If the objective function is truly quadratic, as in the case of fitting a linear regression model with

¹More generally, we could regularize the parameters to be near any specific point in space and, surprisingly, still get a regularization effect, but better results will be obtained for a value closer to the true one, with zero being a default value that makes sense when we do not know if the correct value should be positive or negative. Since it is far more common to regularize the model parameters toward zero, we will focus on this special case in our exposition.

mean squared error, then the approximation is perfect. The approximation \hat{J} is given by

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \mathbf{H}(\boldsymbol{w} - \boldsymbol{w}^*), \quad (7.6)$$

where \mathbf{H} is the Hessian matrix of J with respect to \boldsymbol{w} evaluated at \boldsymbol{w}^* . There is no first-order term in this quadratic approximation, because \boldsymbol{w}^* is defined to be a minimum, where the gradient vanishes. Likewise, because \boldsymbol{w}^* is the location of a minimum of J , we can conclude that \mathbf{H} is positive semidefinite.

The minimum of \hat{J} occurs where its gradient

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \mathbf{H}(\boldsymbol{w} - \boldsymbol{w}^*) \quad (7.7)$$

is equal to $\mathbf{0}$.

To study the effect of weight decay, we modify equation 7.7 by adding the weight decay gradient. We can now solve for the minimum of the regularized version of \hat{J} . We use the variable $\tilde{\boldsymbol{w}}$ to represent the location of the minimum.

$$\alpha \tilde{\boldsymbol{w}} + \mathbf{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = \mathbf{0} \quad (7.8)$$

$$(\mathbf{H} + \alpha \mathbf{I})\tilde{\boldsymbol{w}} = \mathbf{H}\boldsymbol{w}^* \quad (7.9)$$

$$\tilde{\boldsymbol{w}} = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H}\boldsymbol{w}^* \quad (7.10)$$

As α approaches 0, the regularized solution $\tilde{\boldsymbol{w}}$ approaches \boldsymbol{w}^* . But what happens as α grows? Because \mathbf{H} is real and symmetric, we can decompose it into a diagonal matrix Λ and an orthonormal basis of eigenvectors, \mathbf{Q} , such that $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$. Applying the decomposition to equation 7.10, we obtain

$$\tilde{\boldsymbol{w}} = (\mathbf{Q}\Lambda\mathbf{Q}^\top + \alpha \mathbf{I})^{-1} \mathbf{Q}\Lambda\mathbf{Q}^\top \boldsymbol{w}^* \quad (7.11)$$

$$= [\mathbf{Q}(\Lambda + \alpha \mathbf{I})\mathbf{Q}^\top]^{-1} \mathbf{Q}\Lambda\mathbf{Q}^\top \boldsymbol{w}^* \quad (7.12)$$

$$= \mathbf{Q}(\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \boldsymbol{w}^*. \quad (7.13)$$

We see that the effect of weight decay is to rescale \boldsymbol{w}^* along the axes defined by the eigenvectors of \mathbf{H} . Specifically, the component of \boldsymbol{w}^* that is aligned with the i -th eigenvector of \mathbf{H} is rescaled by a factor of $\frac{\lambda_i}{\lambda_i + \alpha}$. (You may wish to review how this kind of scaling works, first explained in figure 2.3).

Along the directions where the eigenvalues of \mathbf{H} are relatively large, for example, where $\lambda_i \gg \alpha$, the effect of regularization is relatively small. Yet components with $\lambda_i \ll \alpha$ will be shrunk to have nearly zero magnitude. This effect is illustrated in figure 7.1.

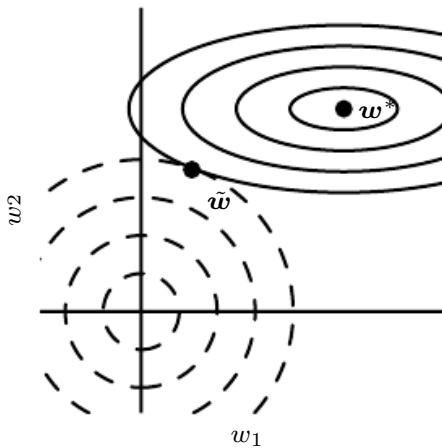


Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal \mathbf{w} . The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\mathbf{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from \mathbf{w}^* . Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function is very sensitive to movements away from \mathbf{w}^* . The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little.

Only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact. In directions that do not contribute to reducing the objective function, a small eigenvalue of the Hessian tells us that movement in this direction will not significantly increase the gradient. Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training.

So far we have discussed weight decay in terms of its effect on the optimization of an abstract, general quadratic cost function. How do these effects relate to machine learning in particular? We can find out by studying linear regression, a model for which the true cost function is quadratic and therefore amenable to the same kind of analysis we have used so far. Applying the analysis again, we will be able to obtain a special case of the same results, but with the solution now phrased in terms of the training data. For linear regression, the cost function is the sum of squared errors:

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}). \quad (7.14)$$

When we add L^2 regularization, the objective function changes to

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\alpha\mathbf{w}^\top \mathbf{w}. \quad (7.15)$$

This changes the normal equations for the solution from

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (7.16)$$

to

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}. \quad (7.17)$$

The matrix $\mathbf{X}^\top \mathbf{X}$ in equation 7.16 is proportional to the covariance matrix $\frac{1}{m} \mathbf{X}^\top \mathbf{X}$. Using L^2 regularization replaces this matrix with $(\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1}$ in equation 7.17. The new matrix is the same as the original one, but with the addition of α to the diagonal. The diagonal entries of this matrix correspond to the variance of each input feature. We can see that L^2 regularization causes the learning algorithm to “perceive” the input \mathbf{X} as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance.

7.1.2 L^1 Regularization

While L^2 weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use L^1 regularization.

Formally, L^1 regularization on the model parameter \mathbf{w} is defined as

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|, \quad (7.18)$$

that is, as the sum of absolute values of the individual parameters.² We will now discuss the effect of L^1 regularization on the simple linear regression model, with no bias parameter, that we studied in our analysis of L^2 regularization. In particular, we are interested in delineating the differences between L^1 and L^2 forms of regularization. As with L^2 weight decay, L^1 weight decay controls the strength of the regularization by scaling the penalty Ω using a positive hyperparameter α . Thus, the regularized objective function $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$ is given by

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\mathbf{w}\|_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.19)$$

with the corresponding gradient (actually, sub gradient)

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{X}, \mathbf{y}; \mathbf{w}), \quad (7.20)$$

where $\text{sign}(\mathbf{w})$ is simply the sign of \mathbf{w} applied element-wise.

By inspecting equation 7.20, we can see immediately that the effect of L^1 regularization is quite different from that of L^2 regularization. Specifically, we can see that the regularization contribution to the gradient no longer scales linearly with each w_i ; instead it is a constant factor with a sign equal to $\text{sign}(w_i)$. One consequence of this form of the gradient is that we will not necessarily see clean algebraic solutions to quadratic approximations of $J(\mathbf{X}, \mathbf{y}; \mathbf{w})$ as we did for L^2 regularization.

Our simple linear model has a quadratic cost function that we can represent via its Taylor series. Alternately, we could imagine that this is a truncated Taylor series approximating the cost function of a more sophisticated model. The gradient in this setting is given by

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.21)$$

where, again, \mathbf{H} is the Hessian matrix of J with respect to \mathbf{w} evaluated at \mathbf{w}^* .

Because the L^1 penalty does not admit clean algebraic expressions in the case of a fully general Hessian, we will also make the further simplifying assumption that the Hessian is diagonal, $\mathbf{H} = \text{diag}([H_{1,1}, \dots, H_{n,n}])$, where each $H_{i,i} > 0$.

²As with L^2 regularization, we could regularize the parameters toward a value that is not zero, but instead toward some parameter value $\mathbf{w}^{(o)}$. In that case the L^1 regularization would introduce the term $\Omega(\boldsymbol{\theta}) = \|\mathbf{w} - \mathbf{w}^{(o)}\|_1 = \sum_i |w_i - w_i^{(o)}|$.

This assumption holds if the data for the linear regression problem has been preprocessed to remove all correlation between the input features, which may be accomplished using PCA.

Our quadratic approximation of the L^1 regularized objective function decomposes into a sum over the parameters:

$$\hat{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}^*; \mathbf{X}, \mathbf{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (\mathbf{w}_i - \mathbf{w}_i^*)^2 + \alpha |\mathbf{w}_i| \right]. \quad (7.22)$$

The problem of minimizing this approximate cost function has an analytical solution (for each dimension i), with the following form:

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}. \quad (7.23)$$

Consider the situation where $w_i^* > 0$ for all i . There are two possible outcomes:

1. The case where $w_i^* \leq \frac{\alpha}{H_{i,i}}$. Here the optimal value of w_i under the regularized objective is simply $w_i = 0$. This occurs because the contribution of $J(\mathbf{w}; \mathbf{X}, \mathbf{y})$ to the regularized objective $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$ is overwhelmed—in direction i —by the L^1 regularization, which pushes the value of w_i to zero.
2. The case where $w_i^* > \frac{\alpha}{H_{i,i}}$. In this case, the regularization does not move the optimal value of w_i to zero but instead just shifts it in that direction by a distance equal to $\frac{\alpha}{H_{i,i}}$.

A similar process happens when $w_i^* < 0$, but with the L^1 penalty making w_i less negative by $\frac{\alpha}{H_{i,i}}$, or 0.

In comparison to L^2 regularization, L^1 regularization results in a solution that is more **sparse**. Sparsity in this context refers to the fact that some parameters have an optimal value of zero. The sparsity of L^1 regularization is a qualitatively different behavior than arises with L^2 regularization. Equation 7.13 gave the solution \tilde{w} for L^2 regularization. If we revisit that equation using the assumption of a diagonal and positive definite Hessian \mathbf{H} that we introduced for our analysis of L^1 regularization, we find that $\tilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^*$. If w_i^* was nonzero, then \tilde{w}_i remains nonzero. This demonstrates that L^2 regularization does not cause the parameters to become sparse, while L^1 regularization may do so for large enough α .

The sparsity property induced by L^1 regularization has been used extensively as a **feature selection** mechanism. Feature selection simplifies a machine learning problem by choosing which subset of the available features should be used. In

particular, the well known LASSO (Tibshirani, 1995) (least absolute shrinkage and selection operator) model integrates an L^1 penalty with a linear model and a least-squares cost function. The L^1 penalty causes a subset of the weights to become zero, suggesting that the corresponding features may safely be discarded.

In section 5.6.1, we saw that many regularization strategies can be interpreted as MAP Bayesian inference, and that in particular, L^2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on the weights. For L^1 regularization, the penalty $\alpha\Omega(\mathbf{w}) = \alpha\sum_i |w_i|$ used to regularize a cost function is equivalent to the log-prior term that is maximized by MAP Bayesian inference when the prior is an isotropic Laplace distribution (equation 3.26) over $\mathbf{w} \in \mathbb{R}^n$:

$$\log p(\mathbf{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha\|\mathbf{w}\|_1 + n \log \alpha - n \log 2. \quad (7.24)$$

From the point of view of learning via maximization with respect to \mathbf{w} , we can ignore the $\log \alpha - \log 2$ terms because they do not depend on \mathbf{w} .

7.2 Norm Penalties as Constrained Optimization

Consider the cost function regularized by a parameter norm penalty:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}). \quad (7.25)$$

Recall from section 4.4 that we can minimize a function subject to constraints by constructing a generalized Lagrange function, consisting of the original objective function plus a set of penalties. Each penalty is a product between a coefficient, called a Karush–Kuhn–Tucker (KKT) multiplier, and a function representing whether the constraint is satisfied. If we wanted to constrain $\Omega(\boldsymbol{\theta})$ to be less than some constant k , we could construct a generalized Lagrange function

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k). \quad (7.26)$$

The solution to the constrained problem is given by

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha). \quad (7.27)$$

As described in section 4.4, solving this problem requires modifying both $\boldsymbol{\theta}$ and α . Section 4.5 provides a worked example of linear regression with an L^2 constraint. Many different procedures are possible—some may use gradient descent,

while others may use analytical solutions for where the gradient is zero—but in all procedures α must increase whenever $\Omega(\boldsymbol{\theta}) > k$ and decrease whenever $\Omega(\boldsymbol{\theta}) < k$. All positive α encourage $\Omega(\boldsymbol{\theta})$ to shrink. The optimal value α^* will encourage $\Omega(\boldsymbol{\theta})$ to shrink, but not so strongly to make $\Omega(\boldsymbol{\theta})$ become less than k .

To gain some insight into the effect of the constraint, we can fix α^* and view the problem as just a function of $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \arg \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha^* \Omega(\boldsymbol{\theta}). \quad (7.28)$$

This is exactly the same as the regularized training problem of minimizing \tilde{J} . We can thus think of a parameter norm penalty as imposing a constraint on the weights. If Ω is the L^2 norm, then the weights are constrained to lie in an L^2 ball. If Ω is the L^1 norm, then the weights are constrained to lie in a region of limited L^1 norm. Usually we do not know the size of the constraint region that we impose by using weight decay with coefficient α^* because the value of α^* does not directly tell us the value of k . In principle, one can solve for k , but the relationship between k and α^* depends on the form of J . While we do not know the exact size of the constraint region, we can control it roughly by increasing or decreasing α in order to grow or shrink the constraint region. Larger α will result in a smaller constraint region. Smaller α will result in a larger constraint region.

Sometimes we may wish to use explicit constraints rather than penalties. As described in section 4.4, we can modify algorithms such as stochastic gradient descent to take a step downhill on $J(\boldsymbol{\theta})$ and then project $\boldsymbol{\theta}$ back to the nearest point that satisfies $\Omega(\boldsymbol{\theta}) < k$. This can be useful if we have an idea of what value of k is appropriate and do not want to spend time searching for the value of α that corresponds to this k .

Another reason to use explicit constraints and reprojection rather than enforcing constraints with penalties is that penalties can cause nonconvex optimization procedures to get stuck in local minima corresponding to small $\boldsymbol{\theta}$. When training neural networks, this usually manifests as neural networks that train with several “dead units.” These are units that do not contribute much to the behavior of the function learned by the network because the weights going into or out of them are all very small. When training with a penalty on the norm of the weights, these configurations can be locally optimal, even if it is possible to significantly reduce J by making the weights larger. Explicit constraints implemented by reprojection can work much better in these cases because they do not encourage the weights to approach the origin. Explicit constraints implemented by reprojection have an effect only when the weights become large and attempt to leave the constraint region.

Finally, explicit constraints with reprojection can be useful because they impose some stability on the optimization procedure. When using high learning rates, it is possible to encounter a positive feedback loop in which large weights induce large gradients, which then induce a large update to the weights. If these updates consistently increase the size of the weights, then θ rapidly moves away from the origin until numerical overflow occurs. Explicit constraints with reprojection prevent this feedback loop from continuing to increase the magnitude of the weights without bound. Hinton *et al.* (2012c) recommend using constraints combined with a high learning rate to enable rapid exploration of parameter space while maintaining some stability.

In particular, Hinton *et al.* (2012c) recommend a strategy introduced by Srebro and Shraibman (2005): constraining the norm of each *column* of the weight matrix of a neural net layer, rather than constraining the Frobenius norm of the entire weight matrix. Constraining the norm of each column separately prevents any one hidden unit from having very large weights. If we converted this constraint into a penalty in a Lagrange function, it would be similar to L^2 weight decay but with a separate KKT multiplier for the weights of each hidden unit. Each of these KKT multipliers would be dynamically updated separately to make each hidden unit obey the constraint. In practice, column norm limitation is always implemented as an explicit constraint with reprojection.

7.3 Regularization and Under-Constrained Problems

In some cases, regularization is necessary for machine learning problems to be properly defined. Many linear models in machine learning, including linear regression and PCA, depend on inverting the matrix $\mathbf{X}^\top \mathbf{X}$. This is not possible when $\mathbf{X}^\top \mathbf{X}$ is singular. This matrix can be singular whenever the data-generating distribution truly has no variance in some direction, or when no variance is *observed* in some direction because there are fewer examples (rows of \mathbf{X}) than input features (columns of \mathbf{X}). In this case, many forms of regularization correspond to inverting $\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I}$ instead. This regularized matrix is guaranteed to be invertible.

These linear problems have closed form solutions when the relevant matrix is invertible. It is also possible for a problem with no closed form solution to be underdetermined. An example is logistic regression applied to a problem where the classes are linearly separable. If a weight vector \mathbf{w} is able to achieve perfect classification, then $2\mathbf{w}$ will also achieve perfect classification and higher likelihood. An iterative optimization procedure like stochastic gradient descent will continually increase the magnitude of \mathbf{w} and, in theory, will never halt. In practice, a numerical

implementation of gradient descent will eventually reach sufficiently large weights to cause numerical overflow, at which point its behavior will depend on how the programmer has decided to handle values that are not real numbers.

Most forms of regularization are able to guarantee the convergence of iterative methods applied to underdetermined problems. For example, weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient.

The idea of using regularization to solve underdetermined problems extends beyond machine learning. The same idea is useful for several basic linear algebra problems.

As we saw in section 2.9, we can solve underdetermined linear equations using the Moore-Penrose pseudoinverse. Recall that one definition of the pseudoinverse \mathbf{X}^+ of a matrix \mathbf{X} is

$$\mathbf{X}^+ = \lim_{\alpha \searrow 0} (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top. \quad (7.29)$$

We can now recognize equation 7.29 as performing linear regression with weight decay. Specifically, equation 7.29 is the limit of equation 7.17 as the regularization coefficient shrinks to zero. We can thus interpret the pseudoinverse as stabilizing underdetermined problems using regularization.

7.4 Dataset Augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.

This approach is easiest for classification. A classifier needs to take a complicated, high-dimensional input \mathbf{x} and summarize it with a single category identity y . This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new (\mathbf{x}, y) pairs easily just by transforming the \mathbf{x} inputs in our training set.

This approach is not as readily applicable to many other tasks. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved the density estimation problem.

Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include

an enormous range of factors of variation, many of which can be easily simulated. Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant by using the convolution and pooling techniques described in chapter 9. Many other operations, such as rotating the image or scaling the image, have also proved quite effective.

One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between “b” and “d” and the difference between “6” and “9,” so horizontal flips and 180° rotations are not appropriate ways of augmenting datasets for these tasks.

There are also transformations that we would like our classifiers to be invariant to but that are not easy to perform. For example, out-of-plane rotation cannot be implemented as a simple geometric operation on the input pixels.

Dataset augmentation is effective for speech recognition tasks as well (Jaitly and Hinton, 2013).

Injecting noise in the input to a neural network (Sietsma and Dow, 1991) can also be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. Neural networks prove not to be very robust to noise, however (Tang and Eliasmith, 2010). One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. Input noise injection is part of some unsupervised learning algorithms, such as the denoising autoencoder (Vincent *et al.*, 2008). Noise injection also works when the noise is applied to the hidden units, which can be seen as doing dataset augmentation at multiple levels of abstraction. Poole *et al.* (2014) recently showed that this approach can be highly effective provided that the magnitude of the noise is carefully tuned. Dropout, a powerful regularization strategy that will be described in section 7.12, can be seen as a process of constructing new inputs by *multiplying* by noise.

When comparing machine learning benchmark results, taking the effect of dataset augmentation into account is important. Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique. To compare the performance of one machine learning algorithm to another, it is necessary to perform controlled experiments. When comparing machine learning algorithm A and machine learning algorithm B, make sure that both algorithms are evaluated using the same hand-designed dataset augmentation schemes. Suppose that algorithm A performs poorly with no dataset

augmentation, and algorithm B performs well when combined with numerous synthetic transformations of the input. In such a case the synthetic transformations likely caused the improved performance, rather than the use of machine learning algorithm B. Sometimes deciding whether an experiment has been properly controlled requires subjective judgment. For example, machine learning algorithms that inject noise into the input are performing a form of dataset augmentation. Usually, operations that are generally applicable (such as adding Gaussian noise to the input) are considered part of the machine learning algorithm, while operations that are specific to one application domain (such as randomly cropping an image) are considered to be separate preprocessing steps.

7.5 Noise Robustness

Section 7.4 has motivated the use of noise applied to the inputs as a dataset augmentation strategy. For some models, the addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights (Bishop, 1995a,b). In the general case, it is important to remember that noise injection can be much more powerful than simply shrinking the parameters, especially when the noise is added to the hidden units. Noise applied to the hidden units is such an important topic that it merits its own separate discussion; the dropout algorithm described in section 7.12 is the main development of that approach.

Another way that noise has been used in the service of regularizing models is by adding it to the weights. This technique has been used primarily in the context of recurrent neural networks (Jim *et al.*, 1996; Graves, 2011). This can be interpreted as a stochastic implementation of Bayesian inference over the weights. The Bayesian treatment of learning would consider the model weights to be uncertain and representable via a probability distribution that reflects this uncertainty. Adding noise to the weights is a practical, stochastic way to reflect this uncertainty.

Noise applied to the weights can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization, encouraging stability of the function to be learned. Consider the regression setting, where we wish to train a function $\hat{y}(\mathbf{x})$ that maps a set of features \mathbf{x} to a scalar using the least-squares cost function between the model predictions $\hat{y}(\mathbf{x})$ and the true values y :

$$J = \mathbb{E}_{p(x,y)} [(\hat{y}(\mathbf{x}) - y)^2]. \quad (7.30)$$

The training set consists of m labeled examples $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})\}$.

We now assume that with each input presentation we also include a random perturbation $\epsilon_{\mathbf{W}} \sim \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \eta \mathbf{I})$ of the network weights. Let us imagine that we have a standard l -layer MLP. We denote the perturbed model as $\hat{y}_{\epsilon_{\mathbf{W}}}(\mathbf{x})$. Despite the injection of noise, we are still interested in minimizing the squared error of the output of the network. The objective function thus becomes

$$\tilde{J}_{\mathbf{W}} = \mathbb{E}_{p(\mathbf{x}, y, \epsilon_{\mathbf{W}})} \left[(\hat{y}_{\epsilon_{\mathbf{W}}}(\mathbf{x}) - y)^2 \right] \quad (7.31)$$

$$= \mathbb{E}_{p(\mathbf{x}, y, \epsilon_{\mathbf{W}})} \left[\hat{y}_{\epsilon_{\mathbf{W}}}^2(\mathbf{x}) - 2y\hat{y}_{\epsilon_{\mathbf{W}}}(\mathbf{x}) + y^2 \right]. \quad (7.32)$$

For small η , the minimization of J with added weight noise (with covariance $\eta \mathbf{I}$) is equivalent to minimization of J with an additional regularization term: $\eta \mathbb{E}_{p(\mathbf{x}, y)} [\|\nabla_{\mathbf{W}} \hat{y}(\mathbf{x})\|^2]$. This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output. In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions (Hochreiter and Schmidhuber, 1995). In the simplified case of linear regression (where, for instance, $\hat{y}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$), this regularization term collapses into $\eta \mathbb{E}_{p(\mathbf{x})} [\|\mathbf{x}\|^2]$, which is not a function of parameters and therefore does not contribute to the gradient of $\tilde{J}_{\mathbf{W}}$ with respect to the model parameters.

7.5.1 Injecting Noise at the Output Targets

Most datasets have some number of mistakes in the y labels. It can be harmful to maximize $\log p(y | \mathbf{x})$ when y is a mistake. One way to prevent this is to explicitly model the noise on the labels. For example, we can assume that for some small constant ϵ , the training set label y is correct with probability $1 - \epsilon$, and otherwise any of the other possible labels might be correct. This assumption is easy to incorporate into the cost function analytically, rather than by explicitly drawing noise samples. For example, **label smoothing** regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1 - \epsilon$, respectively. The standard cross-entropy loss may then be used with these soft targets. Maximum likelihood learning with a softmax classifier and hard targets may actually never converge—the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions forever. It is possible to prevent this scenario using other regularization strategies like weight decay. Label smoothing has the advantage of preventing the pursuit of hard probabilities without discouraging correct classification. This strategy has been used since the 1980s

and continues to be featured prominently in modern neural networks ([Szegedy et al., 2015](#)).

7.6 Semi-Supervised Learning

In the paradigm of semi-supervised learning, both unlabeled examples from $P(\mathbf{x})$ and labeled examples from $P(\mathbf{x}, \mathbf{y})$ are used to estimate $P(\mathbf{y} | \mathbf{x})$ or predict \mathbf{y} from \mathbf{x} .

In the context of deep learning, semi-supervised learning usually refers to learning a representation $\mathbf{h} = f(\mathbf{x})$. The goal is to learn a representation so that examples from the same class have similar representations. Unsupervised learning can provide useful clues for how to group examples in representation space. Examples that cluster tightly in the input space should be mapped to similar representations. A linear classifier in the new space may achieve better generalization in many cases ([Belkin and Niyogi, 2002](#); [Chapelle et al., 2003](#)). A long-standing variant of this approach is the application of principal components analysis as a preprocessing step before applying a classifier (on the projected data).

Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of either $P(\mathbf{x})$ or $P(\mathbf{x}, \mathbf{y})$ shares parameters with a discriminative model of $P(\mathbf{y} | \mathbf{x})$. One can then trade off the supervised criterion $-\log P(\mathbf{y} | \mathbf{x})$ with the unsupervised or generative one (such as $-\log P(\mathbf{x})$ or $-\log P(\mathbf{x}, \mathbf{y})$). The generative criterion then expresses a particular form of prior belief about the solution to the supervised learning problem ([Lasserre et al., 2006](#)), namely that the structure of $P(\mathbf{x})$ is connected to the structure of $P(\mathbf{y} | \mathbf{x})$ in a way that is captured by the shared parametrization. By controlling how much of the generative criterion is included in the total criterion, one can find a better trade-off than with a purely generative or a purely discriminative training criterion ([Lasserre et al., 2006](#); [Larochelle and Bengio, 2008](#)).

[Salakhutdinov and Hinton \(2008\)](#) describe a method for learning the kernel function of a kernel machine used for regression, in which the usage of unlabeled examples for modeling $P(\mathbf{x})$ improves $P(\mathbf{y} | \mathbf{x})$ quite significantly.

See [Chapelle et al. \(2006\)](#) for more information about semi-supervised learning.

7.7 Multitask Learning

Multitask learning (Caruana, 1993) is a way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks. In the same way that additional training examples put more pressure on the parameters of the model toward values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained toward good values (assuming the sharing is justified), often yielding better generalization.

Figure 7.2 illustrates a very common form of multitask learning, in which different supervised tasks (predicting $\mathbf{y}^{(i)}$ given \mathbf{x}) share the same input \mathbf{x} , as well as some intermediate-level representation $\mathbf{h}^{(\text{shared})}$, capturing a common pool of factors. The model can generally be divided into two kinds of parts and associated parameters:

1. Task-specific parameters (which only benefit from the examples of their task to achieve good generalization). These are the upper layers of the neural network in figure 7.2.
2. Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). These are the lower layers of the neural network in figure 7.2.

Improved generalization and generalization error bounds (Baxter, 1995) can be achieved because of the shared parameters, for which statistical strength can be greatly improved (in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models). Of course this will happen only if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.

From the point of view of deep learning, the underlying prior belief is the following: *among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.*

7.8 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but

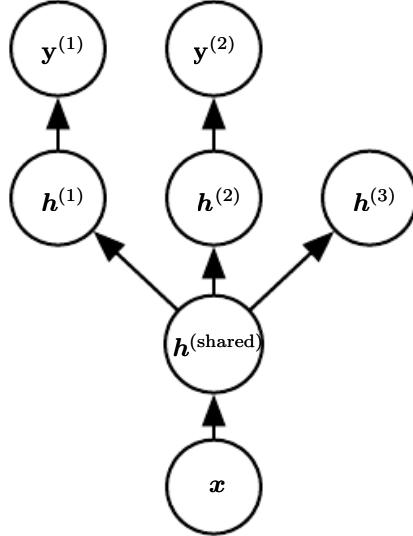


Figure 7.2: Multitask learning can be cast in several ways in deep learning frameworks, and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters (associated respectively with the weights into and from $h^{(1)}$ and $h^{(2)}$) can be learned on top of those yielding a shared representation $h^{(\text{shared})}$. The underlying assumption is that there exists a common pool of factors that explain the variations in the input x , while each task is associated with a subset of these factors. In this example, it is additionally assumed that top-level hidden units $h^{(1)}$ and $h^{(2)}$ are specialized to each task (respectively predicting $y^{(1)}$ and $y^{(2)}$), while some intermediate-level representation $h^{(\text{shared})}$ is shared across all tasks. In the unsupervised learning context, it makes sense for some of the top-level factors to be associated with none of the output tasks ($h^{(3)}$): these are the factors that explain some of the input variations but are not relevant for predicting $y^{(1)}$ or $y^{(2)}$.

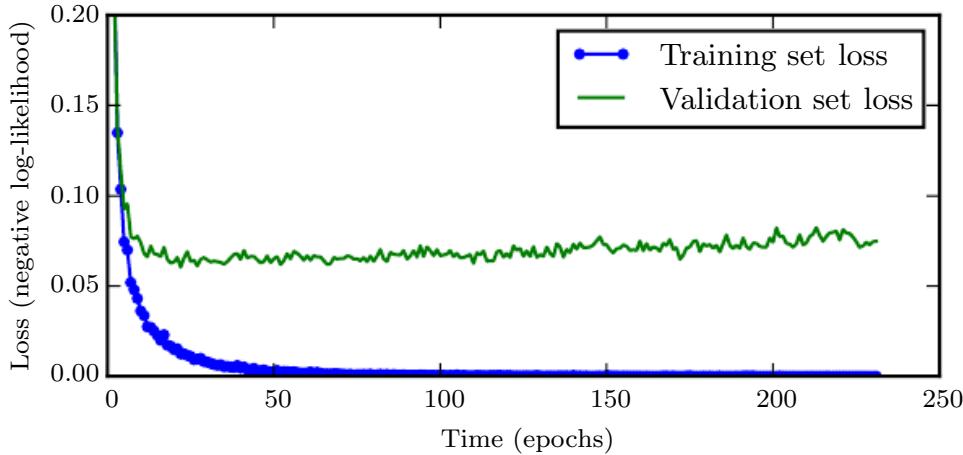


Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or **epochs**). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

validation set error begins to rise again. See figure 7.3 for an example of this behavior, which occurs reliably.

This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This procedure is specified more formally in algorithm 7.1.

This strategy is known as **early stopping**. It is probably the most commonly used form of regularization in deep learning. Its popularity is due to both its effectiveness and its simplicity.

One way to think of early stopping is as a very efficient hyperparameter selection algorithm. In this view, the number of training steps is just another hyperparameter. We can see in figure 7.3 that this hyperparameter has a U-shaped validation set performance curve. Most hyperparameters that control model capacity have such a U-shaped validation set performance curve, as illustrated in figure 5.3. In the case of early stopping, we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set. Most hyperparameters must be

chosen using an expensive guess and check process, where we set a hyperparameter at the start of training, then run training for several steps to see its effect. The “training time” hyperparameter is unique in that by definition, a single run of training tries out many values of the hyperparameter. The only significant cost to choosing this hyperparameter automatically via early stopping is running the validation set evaluation periodically during training. Ideally, this is done in parallel to the training process on a separate machine, separate CPU, or separate GPU from the main training process. If such resources are not available, then the cost of these periodic evaluations may be reduced by using a validation set that is

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.

Let p be the “patience,” the number of times to observe worsening validation set error before giving up.

Let θ_o be the initial parameters.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

while $j < p$ **do**

 Update θ by running the training algorithm for n steps.

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

if $v' < v$ **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

else

$j \leftarrow j + 1$

end if

end while

Best parameters are θ^* , best number of training steps is i^* .

small compared to the training set or by evaluating the validation set error less frequently and obtaining a lower-resolution estimate of the optimal training time.

An additional cost to early stopping is the need to maintain a copy of the best parameters. This cost is generally negligible, because it is acceptable to store these parameters in a slower and larger form of memory (for example, training in GPU memory, but storing the optimal parameters in host memory or on a disk drive). Since the best parameters are written to infrequently and never read during training, these occasional slow writes have little effect on the total training time.

Early stopping is an unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics. This is in contrast to weight decay, where one must be careful not to use too much weight decay and trap the network in a bad local minimum corresponding to a solution with pathologically small weights.

Early stopping may be used either alone or in conjunction with other regularization strategies. Even when using regularization strategies that modify the objective function to encourage better generalization, it is rare for the best generalization to occur at a local minimum of the training objective.

Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed. In the second, extra training step, all the training data is included. There are two basic strategies one can use for this second training procedure.

One strategy (algorithm 7.2) is to initialize the model again and retrain on all the data. In this second training pass, we train for the same number of steps as the early stopping procedure determined was optimal in the first pass. There are some subtleties associated with this procedure. For example, there is not a good way of knowing whether to retrain for the same number of parameter updates or the same number of passes through the dataset. On the second round of training, each pass through the dataset will require more parameter updates because the training set is bigger.

Another strategy for using all the data is to keep the parameters obtained from the first round of training and then *continue* training, but now using all the data. At this stage, we now no longer have a guide for when to stop in terms of a number of steps. Instead, we can monitor the average loss function on the validation set and continue training until it falls below the value of the training set objective at which the early stopping procedure halted. This strategy avoids the high cost of

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ be the training set.

Split $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ into $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$ and $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$ respectively.

Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\mathbf{X}^{(\text{subtrain})}$ and $\mathbf{y}^{(\text{subtrain})}$ for training data and $\mathbf{X}^{(\text{valid})}$ and $\mathbf{y}^{(\text{valid})}$ for validation data. This returns i^* , the optimal number of steps.

Set $\boldsymbol{\theta}$ to random values again.

Train on $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ for i^* steps.

Algorithm 7.3 Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

Let $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ be the training set.

Split $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ into $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$ and $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$ respectively.

Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\mathbf{X}^{(\text{subtrain})}$ and $\mathbf{y}^{(\text{subtrain})}$ for training data and $\mathbf{X}^{(\text{valid})}$ and $\mathbf{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.

$\epsilon \leftarrow J(\boldsymbol{\theta}, \mathbf{X}^{(\text{subtrain})}, \mathbf{y}^{(\text{subtrain})})$

while $J(\boldsymbol{\theta}, \mathbf{X}^{(\text{valid})}, \mathbf{y}^{(\text{valid})}) > \epsilon$ **do**

 Train on $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ for n steps.

end while

retraining the model from scratch but is not as well behaved. For example, the objective on the validation set may not ever reach the target value, so this strategy is not even guaranteed to terminate. This procedure is presented more formally in algorithm 7.3.

Early stopping is also useful because it reduces the computational cost of the training procedure. Besides the obvious reduction in cost due to limiting the number of training iterations, it also has the benefit of providing regularization without requiring the addition of penalty terms to the cost function or the computation of the gradients of such additional terms.

How early stopping acts as a regularizer: So far we have stated that early stopping *is* a regularization strategy, but we have supported this claim only by showing learning curves where the validation set error has a U-shaped curve. What

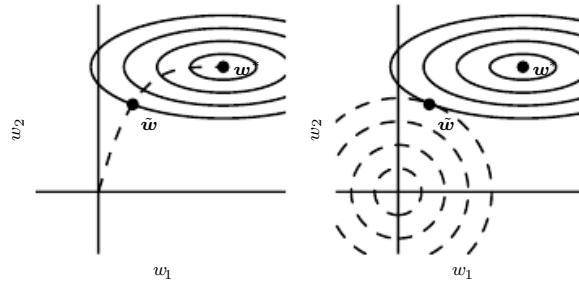


Figure 7.4: An illustration of the effect of early stopping. (*Left*)The solid contour lines indicate the contours of the negative log-likelihood. The dashed line indicates the trajectory taken by SGD beginning from the origin. Rather than stopping at the point \mathbf{w}^* that minimizes the cost, early stopping results in the trajectory stopping at an earlier point $\tilde{\mathbf{w}}$. (*Right*)An illustration of the effect of L^2 regularization for comparison. The dashed circles indicate the contours of the L^2 penalty, which causes the minimum of the total cost to lie nearer the origin than the minimum of the unregularized cost.

is the actual mechanism by which early stopping regularizes the model? Bishop (1995a) and Sjöberg and Ljung (1995) argued that early stopping has the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighborhood of the initial parameter value θ_o , as illustrated in figure 7.4. More specifically, imagine taking τ optimization steps (corresponding to τ training iterations) and with learning rate ϵ . We can view the product $\epsilon\tau$ as a measure of effective capacity. Assuming the gradient is bounded, restricting both the number of iterations and the learning rate limits the volume of parameter space reachable from θ_o . In this sense, $\epsilon\tau$ behaves as if it were the reciprocal of the coefficient used for weight decay.

Indeed, we can show how—in the case of a simple linear model with a quadratic error function and simple gradient descent—early stopping is equivalent to L^2 regularization.

To compare with classical L^2 regularization, we examine a simple setting where the only parameters are linear weights ($\theta = \mathbf{w}$). We can model the cost function J with a quadratic approximation in the neighborhood of the empirically optimal value of the weights \mathbf{w}^* :

$$\hat{J}(\theta) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.33)$$

where \mathbf{H} is the Hessian matrix of J with respect to \mathbf{w} evaluated at \mathbf{w}^* . Given the assumption that \mathbf{w}^* is a minimum of $J(\mathbf{w})$, we know that \mathbf{H} is positive semidefinite.

Under a local Taylor series approximation, the gradient is given by

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \quad (7.34)$$

We are going to study the trajectory followed by the parameter vector during training. For simplicity, let us set the initial parameter vector to the origin,³ $\mathbf{w}^{(0)} = \mathbf{0}$. Let us study the approximate behavior of gradient descent on J by analyzing gradient descent on \hat{J} :

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \epsilon \nabla_{\mathbf{w}} \hat{J}(\mathbf{w}^{(\tau-1)}) \quad (7.35)$$

$$= \mathbf{w}^{(\tau-1)} - \epsilon \mathbf{H}(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*), \quad (7.36)$$

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \epsilon \mathbf{H})(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*). \quad (7.37)$$

Let us now rewrite this expression in the space of the eigenvectors of \mathbf{H} , exploiting the eigendecomposition of \mathbf{H} : $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$, where Λ is a diagonal matrix and \mathbf{Q} is an orthonormal basis of eigenvectors.

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \epsilon \mathbf{Q}\Lambda\mathbf{Q}^\top)(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.38)$$

$$\mathbf{Q}^\top(\mathbf{w}^{(\tau)} - \mathbf{w}^*) = (\mathbf{I} - \epsilon \Lambda)\mathbf{Q}^\top(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.39)$$

Assuming that $\mathbf{w}^{(0)} = \mathbf{0}$ and that ϵ is chosen to be small enough to guarantee $|1 - \epsilon \lambda_i| < 1$, the parameter trajectory during training after τ parameter updates is as follows:

$$\mathbf{Q}^\top \mathbf{w}^{(\tau)} = [\mathbf{I} - (\mathbf{I} - \epsilon \Lambda)^\tau] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.40)$$

Now, the expression for $\mathbf{Q}^\top \tilde{\mathbf{w}}$ in equation 7.13 for L^2 regularization can be rearranged as

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = (\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^*, \quad (7.41)$$

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = [\mathbf{I} - (\Lambda + \alpha \mathbf{I})^{-1} \alpha] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.42)$$

Comparing equation 7.40 and equation 7.42, we see that if the hyperparameters ϵ , α , and τ are chosen such that

$$(\mathbf{I} - \epsilon \Lambda)^\tau = (\Lambda + \alpha \mathbf{I})^{-1} \alpha, \quad (7.43)$$

³For neural networks, to obtain symmetry breaking between hidden units, we cannot initialize all the parameters to $\mathbf{0}$, as discussed in section 6.2. However, the argument holds for any other initial value $\mathbf{w}_{(0)}$.

then L^2 regularization and early stopping can be seen as equivalent (at least under the quadratic approximation of the objective function). Going even further, by taking logarithms and using the series expansion for $\log(1 + x)$, we can conclude that if all λ_i are small (that is, $\epsilon\lambda_i \ll 1$ and $\lambda_i/\alpha \ll 1$) then

$$\tau \approx \frac{1}{\epsilon\alpha}, \quad (7.44)$$

$$\alpha \approx \frac{1}{\tau\epsilon}. \quad (7.45)$$

That is, under these assumptions, the number of training iterations τ plays a role inversely proportional to the L^2 regularization parameter, and the inverse of $\tau\epsilon$ plays the role of the weight decay coefficient.

Parameter values corresponding to directions of significant curvature (of the objective function) are regularized less than directions of less curvature. Of course, in the context of early stopping, this really means that parameters that correspond to directions of significant curvature tend to learn early relative to parameters corresponding to directions of less curvature.

The derivations in this section have shown that a trajectory of length τ ends at a point that corresponds to a minimum of the L^2 -regularized objective. Early stopping is of course more than the mere restriction of the trajectory length; instead, early stopping typically involves monitoring the validation set error in order to stop the trajectory at a particularly good point in space. Early stopping therefore has the advantage over weight decay in that it automatically determines the correct amount of regularization while weight decay requires many training experiments with different values of its hyperparameter.

7.9 Parameter Tying and Parameter Sharing

Thus far, in this chapter, when we have discussed adding constraints or penalties to the parameters, we have always done so with respect to a fixed region or point. For example, L^2 regularization (or weight decay) penalizes model parameters for deviating from the fixed value of zero. Sometimes, however, we may need other ways to express our prior knowledge about suitable values of the model parameters. Sometimes we might not know precisely what values the parameters should take, but we know, from knowledge of the domain and model architecture, that there should be some dependencies between the model parameters.

A common type of dependency that we often want to express is that certain parameters should be close to one another. Consider the following scenario:

we have two models performing the same classification task (with the same set of classes) but with somewhat different input distributions. Formally, we have model A with parameters $\mathbf{w}^{(A)}$ and model B with parameters $\mathbf{w}^{(B)}$. The two models map the input to two different but related outputs: $\hat{y}^{(A)} = f(\mathbf{w}^{(A)}, \mathbf{x})$ and $\hat{y}^{(B)} = g(\mathbf{w}^{(B)}, \mathbf{x})$.

Let us imagine that the tasks are similar enough (perhaps with similar input and output distributions) that we believe the model parameters should be close to each other: $\forall i, w_i^{(A)}$ should be close to $w_i^{(B)}$. We can leverage this information through regularization. Specifically, we can use a parameter norm penalty of the form $\Omega(\mathbf{w}^{(A)}, \mathbf{w}^{(B)}) = \|\mathbf{w}^{(A)} - \mathbf{w}^{(B)}\|_2^2$. Here we used an L^2 penalty, but other choices are also possible.

This kind of approach was proposed by Lasserre *et al.* (2006), who regularized the parameters of one model, trained as a classifier in a supervised paradigm, to be close to the parameters of another model, trained in an unsupervised paradigm (to capture the distribution of the observed input data). The architectures were constructed such that many of the parameters in the classifier model could be paired to corresponding parameters in the unsupervised model.

While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: *to force sets of parameters to be equal*. This method of regularization is often referred to as **parameter sharing**, because we interpret the various models or model components as sharing a unique set of parameters. A significant advantage of parameter sharing over regularizing the parameters to be close (via a norm penalty) is that only a subset of the parameters (the unique set) needs to be stored in memory. In certain models—such as the convolutional neural network—this can lead to significant reduction in the memory footprint of the model.

7.9.1 Convolutional Neural Networks

By far the most popular and extensive use of parameter sharing occurs in **convolutional neural networks** (CNNs) applied to computer vision.

Natural images have many statistical properties that are invariant to translation. For example, a photo of a cat remains a photo of a cat if it is translated one pixel to the right. CNNs take this property into account by sharing parameters across multiple image locations. The same feature (a hidden unit with the same weights) is computed over different locations in the input. This means that we can find a cat with the same cat detector whether the cat appears at column i or column $i + 1$ in the image.

Parameter sharing has enabled CNNs to dramatically lower the number of unique model parameters and to significantly increase network sizes without requiring a corresponding increase in training data. It remains one of the best examples of how to effectively incorporate domain knowledge into the network architecture.

CNNs are discussed in more detail in chapter 9.

7.10 Sparse Representations

Weight decay acts by placing a penalty directly on the model parameters. Another strategy is to place a penalty on the activations of the units in a neural network, encouraging their activations to be sparse. This indirectly imposes a complicated penalty on the model parameters.

We have already discussed (in section 7.1.2) how L^1 penalization induces a sparse parametrization—meaning that many of the parameters become zero (or close to zero). Representational sparsity, on the other hand, describes a representation where many of the elements of the representation are zero (or close to zero). A simplified view of this distinction can be illustrated in the context of linear regression:

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \quad (7.46)$$

$$\mathbf{y} \in \mathbb{R}^m \qquad \mathbf{A} \in \mathbb{R}^{m \times n} \qquad \mathbf{x} \in \mathbb{R}^n$$

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \quad (7.47)$$

$$\mathbf{y} \in \mathbb{R}^m \qquad \mathbf{B} \in \mathbb{R}^{m \times n} \qquad \mathbf{h} \in \mathbb{R}^n$$

In the first expression, we have an example of a sparsely parametrized linear regression model. In the second, we have linear regression with a sparse representa-

tion \mathbf{h} of the data \mathbf{x} . That is, \mathbf{h} is a function of \mathbf{x} that, in some sense, represents the information present in \mathbf{x} , but does so with a sparse vector.

Representational regularization is accomplished by the same sorts of mechanisms that we have used in parameter regularization.

Norm penalty regularization of representations is performed by adding to the loss function J a norm penalty on the *representation*. This penalty is denoted $\Omega(\mathbf{h})$. As before, we denote the regularized loss function by \tilde{J} :

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\mathbf{h}), \quad (7.48)$$

where $\alpha \in [0, \infty)$ weights the relative contribution of the norm penalty term, with larger values of α corresponding to more regularization.

Just as an L^1 penalty on the parameters induces parameter sparsity, an L^1 penalty on the elements of the representation induces representational sparsity: $\Omega(\mathbf{h}) = \|\mathbf{h}\|_1 = \sum_i |h_i|$. Of course, the L^1 penalty is only one choice of penalty that can result in a sparse representation. Others include the penalty derived from a Student t prior on the representation (Olshausen and Field, 1996; Bergstra, 2011) and KL divergence penalties (Larochelle and Bengio, 2008), which are especially useful for representations with elements constrained to lie on the unit interval. Lee *et al.* (2008) and Goodfellow *et al.* (2009) both provide examples of strategies based on regularizing the average activation across several examples, $\frac{1}{m} \sum_i \mathbf{h}^{(i)}$, to be near some target value, such as a vector with .01 for each entry.

Other approaches obtain representational sparsity with a hard constraint on the activation values. For example, **orthogonal matching pursuit** (Pati *et al.*, 1993) encodes an input \mathbf{x} with the representation \mathbf{h} that solves the constrained optimization problem

$$\arg \min_{\mathbf{h}, \|\mathbf{h}\|_0 \leq k} \|\mathbf{x} - \mathbf{W}\mathbf{h}\|^2, \quad (7.49)$$

where $\|\mathbf{h}\|_0$ is the number of nonzero entries of \mathbf{h} . This problem can be solved efficiently when \mathbf{W} is constrained to be orthogonal. This method is often called OMP- k , with the value of k specified to indicate the number of nonzero features allowed. Coates and Ng (2011) demonstrated that OMP-1 can be a very effective feature extractor for deep architectures.

Essentially any model that has hidden units can be made sparse. Throughout this book, we see many examples of sparsity regularization used in various contexts.

7.11 Bagging and Other Ensemble Methods

Bagging (short for **bootstrap aggregating**) is a technique for reducing generalization error by combining several models (Breiman, 1994). The idea is to train several different models separately, then have all the models vote on the output for test examples. This is an example of a general strategy in machine learning called **model averaging**. Techniques employing this strategy are known as **ensemble methods**.

The reason that model averaging works is that different models will usually not make all the same errors on the test set.

Consider for example a set of k regression models. Suppose that each model makes an error ϵ_i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = v$ and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = c$. Then the error made by the average prediction of all the ensemble models is $\frac{1}{k} \sum_i \epsilon_i$. The expected squared error of the ensemble predictor is

$$\mathbb{E} \left[\left(\frac{1}{k} \sum_i \epsilon_i \right)^2 \right] = \frac{1}{k^2} \mathbb{E} \left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j \right) \right], \quad (7.50)$$

$$= \frac{1}{k} v + \frac{k-1}{k} c. \quad (7.51)$$

In the case where the errors are perfectly correlated and $c = v$, the mean squared error reduces to v , so the model averaging does not help at all. In the case where the errors are perfectly uncorrelated and $c = 0$, the expected squared error of the ensemble is only $\frac{1}{k} v$. This means that the expected squared error of the ensemble is inversely proportional to the ensemble size. In other words, on average, the ensemble will perform at least as well as any of its members, and if the members make independent errors, the ensemble will perform significantly better than its members.

Different ensemble methods construct the ensemble of models in different ways. For example, each member of the ensemble could be formed by training a completely different kind of model using a different algorithm or objective function. Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times.

Specifically, bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means that, with high probability, each dataset is missing some of the examples from the

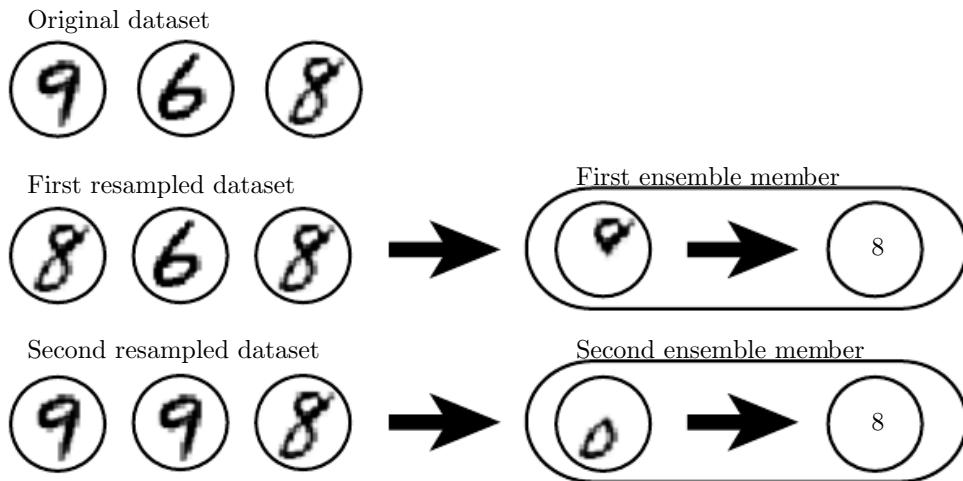


Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an 8 detector on the dataset depicted above, containing an 8, a 6 and a 9. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the 9 and repeats the 8. On this dataset, the detector learns that a loop on top of the digit corresponds to an 8. On the second dataset, we repeat the 9 and omit the 6. In this case, the detector learns that a loop on the bottom of the digit corresponds to an 8. Each of these individual classification rules is brittle, but if we average their output, then the detector is robust, achieving maximal confidence only when both loops of the 8 are present.

original dataset and contains several duplicate examples⁴. Model i is then trained on dataset i . The differences between which examples are included in each dataset result in differences between the trained models. See figure 7.5 for an example.

Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all the models are trained on the same dataset. Differences in random initialization, in random selection of minibatches, in hyperparameters, or in outcomes of nondeterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

Model averaging is an extremely powerful and reliable method for reducing generalization error. Its use is usually discouraged when benchmarking algorithms for scientific papers, because any machine learning algorithm can benefit substantially from model averaging at the price of increased computation and memory.

⁴ When both the original and the resampled dataset contain m examples, the exact proportion of examples missing from the new dataset is $(1 - \frac{1}{m})^m$. This is the chance that a particular example is not chosen among the m possible source examples for all m draws used to create the new dataset. As m approaches infinity this quantity converges to $\frac{1}{e}$, which is slightly larger than $\frac{1}{3}$.

For this reason, benchmark comparisons are usually made using a single model.

Machine learning contests are usually won by methods using model averaging over dozens of models. A recent prominent example is the Netflix Grand Prize (Koren, 2009).

Not all techniques for constructing ensembles are designed to make the ensemble more regularized than the individual models. For example, a technique called **boosting** (Freund and Schapire, 1996b,a) constructs an ensemble with higher capacity than the individual models. Boosting has been applied to build ensembles of neural networks (Schwenk and Bengio, 1998) by incrementally adding neural networks to the ensemble. Boosting has also been applied interpreting an individual neural network as an ensemble (Bengio *et al.*, 2006a), incrementally adding hidden units to the network.

7.12 Dropout

Dropout (Srivastava *et al.*, 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks. Bagging involves training multiple models and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. It is common to use ensembles of five to ten neural networks—Szegedy *et al.* (2014a) used six to win the ILSVRC—but more than this rapidly becomes unwieldy. Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Specifically, dropout trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network, as illustrated in figure 7.6. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero. This procedure requires some slight modification for models such as radial basis function networks, which take the difference between the unit’s state and some reference value. Here, we present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.

Recall that to learn with bagging, we define k different models, construct k

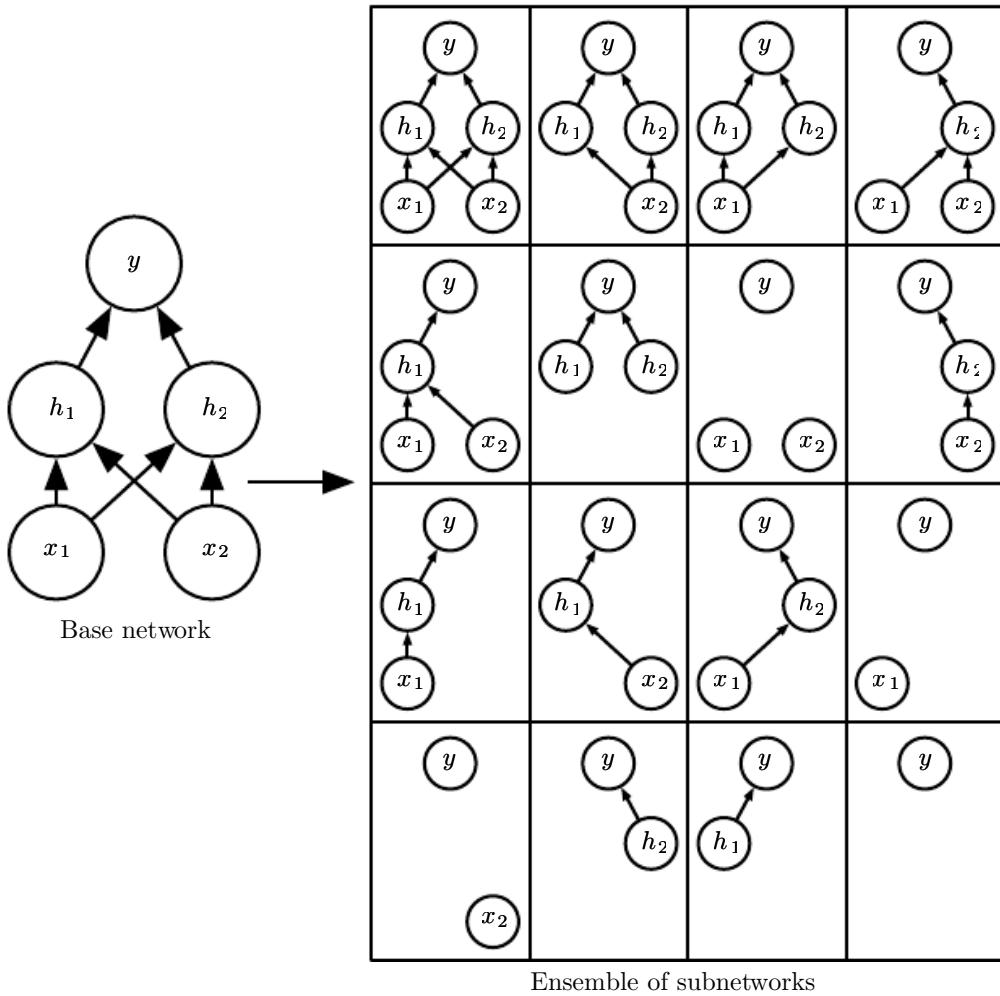


Figure 7.6: Dropout trains an ensemble consisting of all subnetworks that can be constructed by removing nonoutput units from an underlying base network. Here, we begin with a base network with two visible units and two hidden units. There are sixteen possible subsets of these four units. We show all sixteen subnetworks that may be formed by dropping out different subsets of units from the original network. In this small example, a large proportion of the resulting networks have no input units or no path connecting the input to the output. This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.

different datasets by sampling from the training set with replacement, and then train model i on dataset i . Dropout aims to approximate this process, but with an exponentially large number of neural networks. Specifically, to train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent. Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network. The mask for each unit is sampled independently from all the others. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. It is not a function of the current value of the model parameters or the input example. Typically, an input unit is included with probability 0.8, and a hidden unit is included with probability 0.5. We then run forward propagation, back-propagation, and the learning update as usual. Figure 7.7 illustrates how to run forward propagation with dropout.

More formally, suppose that a mask vector μ specifies which units to include, and $J(\theta, \mu)$ defines the cost of the model defined by parameters θ and mask μ . Then dropout training consists of minimizing $\mathbb{E}_\mu J(\theta, \mu)$. The expectation contains exponentially many terms, but we can obtain an unbiased estimate of its gradient by sampling values of μ .

Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory. In the case of bagging, each model is trained to convergence on its respective training set. In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible subnetworks within the lifetime of the universe. Instead, a tiny fraction of the possible subnetworks are each trained for a single step, and the parameter sharing causes the remaining subnetworks to arrive at good settings of the parameters. These are the only differences. Beyond these, dropout follows the bagging algorithm. For example, the training set encountered by each subnetwork is indeed a subset of the original training set sampled with replacement.

To make a prediction, a bagged ensemble must accumulate votes from all its members. We refer to this process as **inference** in this context. So far, our description of bagging and dropout has not required that the model be explicitly probabilistic. Now, we assume that the model’s role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution

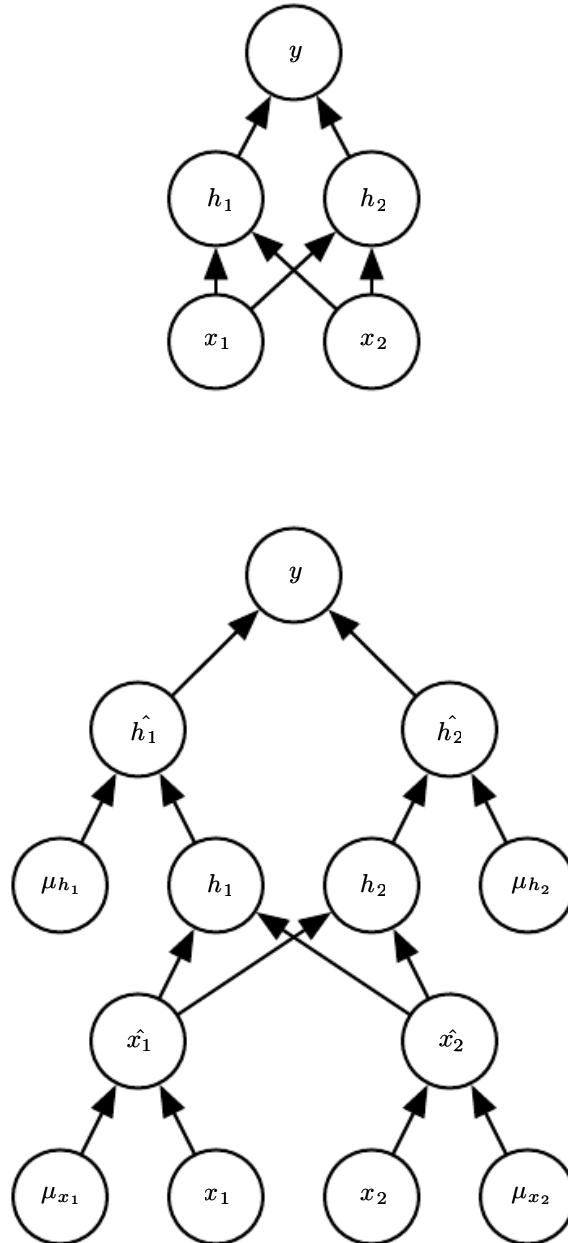


Figure 7.7: An example of forward propagation through a feedforward network using dropout. (Top) In this example, we use a feedforward network with two input units, one hidden layer with two hidden units, and one output unit. (Bottom) To perform forward propagation with dropout, we randomly sample a vector μ with one entry for each input or hidden unit in the network. The entries of μ are binary and are sampled independently from each other. The probability of each entry being 1 is a hyperparameter, usually 0.5 for the hidden layers and 0.8 for the input. Each unit in the network is multiplied by the corresponding mask, and then forward propagation continues through the rest of the network as usual. This is equivalent to randomly selecting one of the sub-networks from figure 7.6 and running forward propagation through it.

$p^{(i)}(y | \mathbf{x})$. The prediction of the ensemble is given by the arithmetic mean of all these distributions,

$$\frac{1}{k} \sum_{i=1}^k p^{(i)}(y | \mathbf{x}). \quad (7.52)$$

In the case of dropout, each submodel defined by mask vector $\boldsymbol{\mu}$ defines a probability distribution $p(y | \mathbf{x}, \boldsymbol{\mu})$. The arithmetic mean over all masks is given by

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y | \mathbf{x}, \boldsymbol{\mu}), \quad (7.53)$$

where $p(\boldsymbol{\mu})$ is the probability distribution that was used to sample $\boldsymbol{\mu}$ at training time.

Because this sum includes an exponential number of terms, it is intractable to evaluate except when the structure of the model permits some form of simplification. So far, deep neural nets are not known to permit any tractable simplification. Instead, we can approximate the inference with sampling, by averaging together the output from many masks. Even 10–20 masks are often sufficient to obtain good performance.

An even better approach, however, allows us to obtain a good approximation to the predictions of the entire ensemble, at the cost of only one forward propagation. To do so, we change to using the geometric mean rather than the arithmetic mean of the ensemble members' predicted distributions. Warde-Farley *et al.* (2014) present arguments and empirical evidence that the geometric mean performs comparably to the arithmetic mean in this context.

The geometric mean of multiple probability distributions is not guaranteed to be a probability distribution. To guarantee that the result is a probability distribution, we impose the requirement that none of the submodels assigns probability 0 to any event, and we renormalize the resulting distribution. The unnormalized probability distribution defined directly by the geometric mean is given by

$$\tilde{p}_{\text{ensemble}}(y | \mathbf{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y | \mathbf{x}, \boldsymbol{\mu})}, \quad (7.54)$$

where d is the number of units that may be dropped. Here we use a uniform distribution over $\boldsymbol{\mu}$ to simplify the presentation, but nonuniform distributions are also possible. To make predictions we must renormalize the ensemble:

$$p_{\text{ensemble}}(y | \mathbf{x}) = \frac{\tilde{p}_{\text{ensemble}}(y | \mathbf{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' | \mathbf{x})}. \quad (7.55)$$

A key insight (Hinton *et al.*, 2012c) involved in dropout is that we can approximate p_{ensemble} by evaluating $p(y | \mathbf{x})$ in one model: the model with all units, but with the weights going out of unit i multiplied by the probability of including unit i . The motivation for this modification is to capture the right expected value of the output from that unit. We call this approach the **weight scaling inference rule**. There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.

Because we usually use an inclusion probability of $\frac{1}{2}$, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training, and then using the model as usual. Another way to achieve the same result is to multiply the states of the units by 2 during training. Either way, the goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.

For many classes of models that do not have nonlinear hidden units, the weight scaling inference rule is exact. For a simple example, consider a softmax regression classifier with n input variables represented by the vector \mathbf{v} :

$$P(y = y | \mathbf{v}) = \text{softmax} \left(\mathbf{W}^\top \mathbf{v} + \mathbf{b} \right)_y. \quad (7.56)$$

We can index into the family of submodels by element-wise multiplication of the input with a binary vector \mathbf{d} :

$$P(y = y | \mathbf{v}; \mathbf{d}) = \text{softmax} \left(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b} \right)_y. \quad (7.57)$$

The ensemble predictor is defined by renormalizing the geometric mean over all ensemble members' predictions:

$$P_{\text{ensemble}}(y = y | \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(y = y | \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(y = y' | \mathbf{v})}, \quad (7.58)$$

where

$$\tilde{P}_{\text{ensemble}}(y = y | \mathbf{v}) = \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} P(y = y | \mathbf{v}; \mathbf{d})}. \quad (7.59)$$

To see that the weight scaling rule is exact, we can simplify $\tilde{P}_{\text{ensemble}}$:

$$\tilde{P}_{\text{ensemble}}(y = y | \mathbf{v}) = \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} P(y = y | \mathbf{v}; \mathbf{d})} \quad (7.60)$$

$$= \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \text{softmax}(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y} \quad (7.61)$$

$$= \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \frac{\exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + b_y)}{\sum_{y'} \exp(\mathbf{W}_{y',:}^\top (\mathbf{d} \odot \mathbf{v}) + b_{y'})}} \quad (7.62)$$

$$= \frac{\sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + b_y)}}{\sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \sum_{y'} \exp(\mathbf{W}_{y',:}^\top (\mathbf{d} \odot \mathbf{v}) + b_{y'})}} \quad (7.63)$$

Because \tilde{P} will be normalized, we can safely ignore multiplication by factors that are constant with respect to y :

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + b_y)} \quad (7.64)$$

$$= \exp\left(\frac{1}{2^n} \sum_{\mathbf{d} \in \{0,1\}^n} \mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + b_y\right) \quad (7.65)$$

$$= \exp\left(\frac{1}{2} \mathbf{W}_{y,:}^\top \mathbf{v} + b_y\right). \quad (7.66)$$

Substituting this back into equation 7.58, we obtain a softmax classifier with weights $\frac{1}{2}\mathbf{W}$.

The weight scaling rule is also exact in other settings, including regression networks with conditionally normal outputs as well as deep networks that have hidden layers without nonlinearities. However, the weight scaling rule is only an approximation for deep models that have nonlinearities. Though the approximation has not been theoretically characterized, it often works well, empirically. [Goodfellow et al. \(2013a\)](#) found experimentally that the weight scaling approximation can work better (in terms of classification accuracy) than Monte Carlo approximations to the ensemble predictor. This held true even when the Monte Carlo approximation was allowed to sample up to 1,000 subnetworks. [Gal and Ghahramani \(2015\)](#) found that some models obtain better classification accuracy using twenty samples and the Monte Carlo approximation. It appears that the optimal choice of inference approximation is problem dependent.

[Srivastava et al. \(2014\)](#) showed that dropout is more effective than other standard computationally inexpensive regularizers, such as weight decay, filter

norm constraints, and sparse activity regularization. Dropout may also be combined with other forms of regularization to yield a further improvement.

One advantage of dropout is that it is very computationally cheap. Using dropout during training requires only $O(n)$ computation per example per update, to generate n random binary numbers and multiply them by the state. Depending on the implementation, it may also require $O(n)$ memory to store these binary numbers until the back-propagation stage. Running inference in the trained model has the same cost per example as if dropout were not used, though we must pay the cost of dividing the weights by 2 once before beginning to run inference on examples.

Another significant advantage of dropout is that it does not significantly limit the type of model or training procedure that can be used. It works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent. This includes feedforward neural networks, probabilistic models such as restricted Boltzmann machines (Srivastava *et al.*, 2014), and recurrent neural networks (Bayer and Osendorfer, 2014; Pascanu *et al.*, 2014a). Many other regularization strategies of comparable power impose more severe restrictions on the architecture of the model.

Though the cost per step of applying dropout to a specific model is negligible, the cost of using dropout in a complete system can be significant. Because dropout is a regularization technique, it reduces the effective capacity of a model. To offset this effect, we must increase the size of the model. Typically the optimal validation set error is much lower when using dropout, but this comes at the cost of a much larger model and many more iterations of the training algorithm. For very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

When extremely few labeled training examples are available, dropout is less effective. Bayesian neural networks (Neal, 1996) outperform dropout on the Alternative Splicing Dataset (Xiong *et al.*, 2011), where fewer than 5,000 examples are available (Srivastava *et al.*, 2014). When additional unlabeled data is available, unsupervised feature learning can gain an advantage over dropout.

Wager *et al.* (2013) showed that, when applied to linear regression, dropout is equivalent to L^2 weight decay, with a different weight decay coefficient for each input feature. The magnitude of each feature's weight decay coefficient is determined by its variance. Similar results hold for other linear models. For deep models, dropout is not equivalent to weight decay.

The stochasticity used while training with dropout is not necessary for the

approach’s success. It is just a means of approximating the sum over all submodels. [Wang and Manning \(2013\)](#) derived analytical approximations to this marginalization. Their approximation, known as **fast dropout**, resulted in faster convergence time due to the reduced stochasticity in the computation of the gradient. This method can also be applied at test time, as a more principled (but also more computationally expensive) approximation to the average over all sub-networks than the weight scaling approximation. Fast dropout has been used to nearly match the performance of standard dropout on small neural network problems, but has not yet yielded a significant improvement or been applied to a large problem.

Just as stochasticity is not necessary to achieve the regularizing effect of dropout, it is also not sufficient. To demonstrate this, [Warde-Farley et al. \(2014\)](#) designed control experiments using a method called **dropout boosting**, which they designed to use exactly the same mask noise as traditional dropout but lack its regularizing effect. Dropout boosting trains the entire ensemble to jointly maximize the log-likelihood on the training set. In the same sense that traditional dropout is analogous to bagging, this approach is analogous to boosting. As intended, experiments with dropout boosting show almost no regularization effect compared to training the entire network as a single model. This demonstrates that the interpretation of dropout as bagging has value beyond the interpretation of dropout as robustness to noise. The regularization effect of the bagged ensemble is only achieved when the stochastically sampled ensemble members are trained to perform well independently of each other.

Dropout has inspired other stochastic approaches to training exponentially large ensembles of models that share weights. DropConnect is a special case of dropout where each product between a single scalar weight and a single hidden unit state is considered a unit that can be dropped ([Wan et al., 2013](#)). Stochastic pooling is a form of randomized pooling (see section 9.3) for building ensembles of convolutional networks, with each convolutional network attending to different spatial locations of each feature map. So far, dropout remains the most widely used implicit ensemble method.

One of the key insights of dropout is that training a network with stochastic behavior and making predictions by averaging over multiple stochastic decisions implements a form of bagging with parameter sharing. Earlier, we described dropout as bagging an ensemble of models formed by including or excluding units. Yet this model averaging strategy does not need to be based on inclusion and exclusion. In principle, any kind of random modification is admissible. In practice, we must choose modification families that neural networks are able to learn to resist. Ideally, we should also use model families that allow a fast approximate

inference rule. We can think of any form of modification parametrized by a vector $\boldsymbol{\mu}$ as training an ensemble consisting of $p(y | \mathbf{x}, \boldsymbol{\mu})$ for all possible values of $\boldsymbol{\mu}$. There is no requirement that $\boldsymbol{\mu}$ have a finite number of values. For example, $\boldsymbol{\mu}$ can be real valued. Srivastava *et al.* (2014) showed that multiplying the weights by $\boldsymbol{\mu} \sim \mathcal{N}(\mathbf{1}, I)$ can outperform dropout based on binary masks. Because $\mathbb{E}[\boldsymbol{\mu}] = \mathbf{1}$, the standard network automatically implements approximate inference in the ensemble, without needing any weight scaling.

So far we have described dropout purely as a means of performing efficient, approximate bagging. Another view of dropout goes further than this. Dropout trains not just a bagged ensemble of models, but an ensemble of models that share hidden units. This means each hidden unit must be able to perform well regardless of which other hidden units are in the model. Hidden units must be prepared to be swapped and interchanged between models. Hinton *et al.* (2012c) were inspired by an idea from biology: sexual reproduction, which involves swapping genes between two different organisms, creates evolutionary pressure for genes to become not just good but readily swapped between different organisms. Such genes and such features are robust to changes in their environment because they are not able to incorrectly adapt to unusual features of any one organism or model. Dropout thus regularizes each hidden unit to be not merely a good feature but a feature that is good in many contexts. Warde-Farley *et al.* (2014) compared dropout training to training of large ensembles and concluded that dropout offers additional improvements to generalization error beyond those obtained by ensembles of independent models.

It is important to understand that a large portion of the power of dropout arises from the fact that the masking noise is applied to the hidden units. This can be seen as a form of highly intelligent, adaptive destruction of the information content of the input rather than destruction of the raw values of the input. For example, if the model learns a hidden unit h_i that detects a face by finding the nose, then dropping h_i corresponds to erasing the information that there is a nose in the image. The model must learn another h_i , that either redundantly encodes the presence of a nose or detects the face by another feature, such as the mouth. Traditional noise injection techniques that add unstructured noise at the input are not able to randomly erase the information about a nose from an image of a face unless the magnitude of the noise is so great that nearly all the information in the image is removed. Destroying extracted features rather than original values allows the destruction process to make use of all the knowledge about the input distribution that the model has acquired so far.

Another important aspect of dropout is that the noise is multiplicative. If the

noise were additive with fixed scale, then a rectified linear hidden unit h_i with added noise ϵ could simply learn to have h_i become very large in order to make the added noise ϵ insignificant by comparison. Multiplicative noise does not allow such a pathological solution to the noise robustness problem.

Another deep learning algorithm, batch normalization, reparametrizes the model in a way that introduces both additive and multiplicative noise on the hidden units at training time. The primary purpose of batch normalization is to improve optimization, but the noise can have a regularizing effect, and sometimes makes dropout unnecessary. Batch normalization is described further in section 8.7.1.

7.13 Adversarial Training

In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set. It is natural therefore to wonder whether these models have obtained a true human-level understanding of these tasks. To probe the level of understanding a network has of the underlying task, we can search for examples that the model misclassifies. Szegedy *et al.* (2014b) found that even neural networks that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally constructed by using an optimization procedure to search for an input \mathbf{x}' near a data point \mathbf{x} such that the model output is very different at \mathbf{x}' . In many cases, \mathbf{x}' can be so similar to \mathbf{x} that a

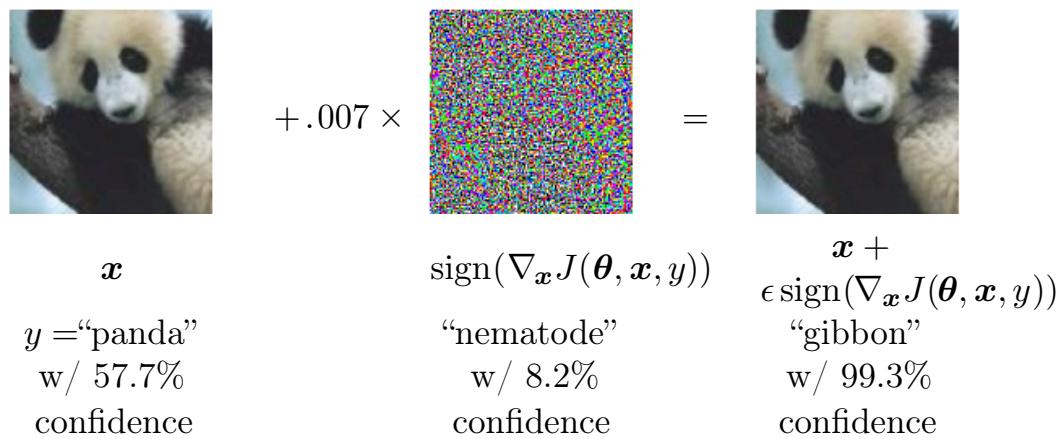


Figure 7.8: A demonstration of adversarial example generation applied to GoogLeNet (Szegedy *et al.*, 2014a) on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet’s classification of the image. Reproduced with permission from Goodfellow *et al.* (2014b).

human observer cannot tell the difference between the original example and the **adversarial example**, but the network can make highly different predictions. See figure 7.8 for an example.

Adversarial examples have many implications, for example, in computer security, that are beyond the scope of this chapter. They are interesting in the context of regularization, however, because one can reduce the error rate on the original i.i.d. test set via **adversarial training**—training on adversarially perturbed examples from the training set (Szegedy *et al.*, 2014b; Goodfellow *et al.*, 2014b).

Goodfellow *et al.* (2014b) showed that one of the primary causes of these adversarial examples is excessive linearity. Neural networks are built out of primarily linear building blocks. In some experiments the overall function they implement proves to be highly linear as a result. These linear functions are easy to optimize. Unfortunately, the value of a linear function can change very rapidly if it has numerous inputs. If we change each input by ϵ , then a linear function with weights \mathbf{w} can change by as much as $\epsilon\|\mathbf{w}\|_1$, which can be a very large amount if \mathbf{w} is highdimensional. Adversarial training discourages this highly sensitive locally linear behavior by encouraging the network to be locally constant in the neighborhood of the training data. This can be seen as a way of explicitly introducing a local constancy prior into supervised neural nets.

Adversarial training helps to illustrate the power of using a large function family in combination with aggressive regularization. Purely linear models, like logistic regression, are not able to resist adversarial examples because they are forced to be linear. Neural networks are able to represent functions that can range from nearly linear to nearly locally constant and thus have the flexibility to capture linear trends in the training data while still learning to resist local perturbation.

Adversarial examples also provide a means of accomplishing semi-supervised learning. At a point \mathbf{x} that is not associated with a label in the dataset, the model itself assigns some label \hat{y} . The model’s label \hat{y} may not be the true label, but if the model is high quality, then \hat{y} has a high probability of providing the true label. We can seek an adversarial example \mathbf{x}' that causes the classifier to output a label y' with $y' \neq \hat{y}$. Adversarial examples generated using not the true label but a label provided by a trained model are called **virtual adversarial examples** (Miyato *et al.*, 2015). The classifier may then be trained to assign the same label to \mathbf{x} and \mathbf{x}' . This encourages the classifier to learn a function that is robust to small changes anywhere along the manifold where the unlabeled data lie. The assumption motivating this approach is that different classes usually lie on disconnected manifolds, and a small perturbation should not be able to jump from one class manifold to another class manifold.

7.14 Tangent Distance, Tangent Prop and Manifold Tangent Classifier

Many machine learning algorithms aim to overcome the curse of dimensionality by assuming that the data lies near a low-dimensional manifold, as described in section 5.11.3.

One of the early attempts to take advantage of the manifold hypothesis is the **tangent distance** algorithm (Simard *et al.*, 1993, 1998). It is a nonparametric nearest neighbor algorithm in which the metric used is not the generic Euclidean distance but one that is derived from knowledge of the manifolds near which probability concentrates. It is assumed that we are trying to classify examples, and that examples on the same manifold share the same category. Since the classifier should be invariant to the local factors of variation that correspond to movement on the manifold, it would make sense to use as nearest neighbor distance between points \mathbf{x}_1 and \mathbf{x}_2 the distance between the manifolds M_1 and M_2 to which they respectively belong. Although that may be computationally difficult (it would require solving an optimization problem, to find the nearest pair of points on M_1 and M_2), a cheap alternative that makes sense locally is to approximate M_i by its tangent plane at \mathbf{x}_i and measure the distance between the two tangents, or between a tangent plane and a point. That can be achieved by solving a low-dimensional linear system (in the dimension of the manifolds). Of course, this algorithm requires one to specify the tangent vectors.

In a related spirit, the **tangent prop** algorithm (Simard *et al.*, 1992) (figure 7.9) trains a neural net classifier with an extra penalty to make each output $f(\mathbf{x})$ of the neural net locally invariant to known factors of variation. These factors of variation correspond to movement along the manifold near which examples of the same class concentrate. Local invariance is achieved by requiring $\nabla_{\mathbf{x}} f(\mathbf{x})$ to be orthogonal to the known manifold tangent vectors $\mathbf{v}^{(i)}$ at \mathbf{x} , or equivalently that the directional derivative of f at \mathbf{x} in the directions $\mathbf{v}^{(i)}$ be small by adding a regularization penalty Ω :

$$\Omega(f) = \sum_i \left((\nabla_{\mathbf{x}} f(\mathbf{x}))^\top \mathbf{v}^{(i)} \right)^2. \quad (7.67)$$

This regularizer can of course be scaled by an appropriate hyperparameter, and for most neural networks, we would need to sum over many outputs rather than the lone output $f(\mathbf{x})$ described here for simplicity. As with the tangent distance algorithm, the tangent vectors are derived a priori, usually from the formal knowledge of the effect of transformations, such as translation, rotation, and scaling in images.

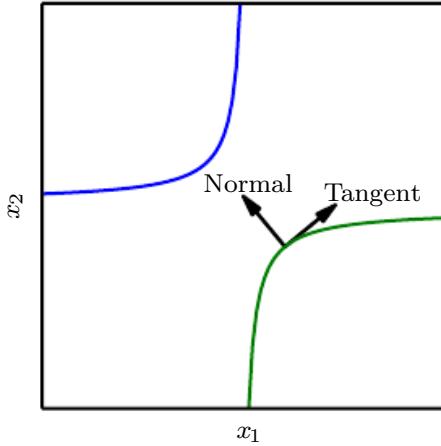


Figure 7.9: Illustration of the main idea of the tangent prop algorithm (Simard *et al.*, 1992) and manifold tangent classifier (Rifai *et al.*, 2011c), which both regularize the classifier output function $f(\mathbf{x})$. Each curve represents the manifold for a different class, illustrated here as a one-dimensional manifold embedded in a two-dimensional space. On one curve, we have chosen a single point and drawn a vector that is tangent to the class manifold (parallel to and touching the manifold) and a vector that is normal to the class manifold (orthogonal to the manifold). In multiple dimensions there may be many tangent directions and many normal directions. We expect the classification function to change rapidly as it moves in the direction normal to the manifold, and not to change as it moves along the class manifold. Both tangent propagation and the manifold tangent classifier regularize $f(\mathbf{x})$ to not change very much as \mathbf{x} moves along the manifold. Tangent propagation requires the user to manually specify functions that compute the tangent directions (such as specifying that small translations of images remain in the same class manifold), while the manifold tangent classifier estimates the manifold tangent directions by training an autoencoder to fit the training data. The use of autoencoders to estimate manifolds is described in chapter 14.

Tangent prop has been used not just for supervised learning (Simard *et al.*, 1992) but also in the context of reinforcement learning (Thrun, 1995).

Tangent propagation is closely related to dataset augmentation. In both cases, the user of the algorithm encodes his or her prior knowledge of the task by specifying a set of transformations that should not alter the output of the network. The difference is that in the case of dataset augmentation, the network is explicitly trained to correctly classify distinct inputs that were created by applying more than an infinitesimal amount of these transformations. Tangent propagation does not require explicitly visiting a new input point. Instead, it analytically regularizes the model to resist perturbation in the directions corresponding to the specified transformation. While this analytical approach is intellectually elegant, it has two major drawbacks. First, it only regularizes the model to resist infinitesimal perturbation. Explicit dataset augmentation confers resistance to larger perturbations. Second, the infinitesimal approach poses difficulties for models based on rectified linear units. These models can only shrink their derivatives by turning units off or shrinking their weights. They are not able to shrink their derivatives by saturating at a high value with large weights, as sigmoid or tanh units can. Dataset augmentation works well with rectified linear units because different subsets of rectified units can activate for different transformed versions of each original input.

Tangent propagation is also related to **double backprop** (Drucker and LeCun, 1992) and adversarial training (Szegedy *et al.*, 2014b; Goodfellow *et al.*, 2014b). Double backprop regularizes the Jacobian to be small, while adversarial training finds inputs near the original inputs and trains the model to produce the same output on these as on the original inputs. Tangent propagation and dataset augmentation using manually specified transformations both require that the model be invariant to certain specified directions of change in the input. Double backprop and adversarial training both require that the model should be invariant to *all* directions of change in the input as long as the change is small. Just as dataset augmentation is the non-infinitesimal version of tangent propagation, adversarial training is the non-infinitesimal version of double backprop.

The manifold tangent classifier (Rifai *et al.*, 2011c), eliminates the need to know the tangent vectors *a priori*. As we will see in chapter 14, autoencoders can estimate the manifold tangent vectors. The manifold tangent classifier makes use of this technique to avoid needing user-specified tangent vectors. As illustrated in figure 14.10, these estimated tangent vectors go beyond the classical invariants that arise out of the geometry of images (such as translation, rotation, and scaling) and include factors that must be learned because they are object-specific (such as

moving body parts). The algorithm proposed with the manifold tangent classifier is therefore simple: (1) use an autoencoder to learn the manifold structure by unsupervised learning, and (2) use these tangents to regularize a neural net classifier as in tangent prop (equation 7.67).

In this chapter, we have described most of the general strategies used to regularize neural networks. Regularization is a central theme of machine learning and as such will be revisited periodically in most of the remaining chapters. Another central theme of machine learning is optimization, described next.

Chapter 8

Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, performing inference in models such as PCA involves solving an optimization problem. We often use analytical optimization to write proofs or design algorithms. Of all the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds of machines to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it. This chapter presents these optimization techniques for neural network training.

If you are unfamiliar with the basic principles of gradient-based optimization, we suggest reviewing chapter 4. That chapter includes a brief overview of numerical optimization in general.

This chapter focuses on one particular case of optimization: finding the parameters θ of a neural network that significantly reduce a cost function $J(\theta)$, which typically includes a performance measure evaluated on the entire training set as well as additional regularization terms.

We begin with a description of how optimization used as a training algorithm for a machine learning task differs from pure optimization. Next, we present several of the concrete challenges that make optimization of neural networks difficult. We then define several practical algorithms, including both optimization algorithms themselves and strategies for initializing the parameters. More advanced algorithms adapt their learning rates during training or leverage information contained in

the second derivatives of the cost function. Finally, we conclude with a review of several optimization strategies that are formed by combining simple optimization algorithms into higher-level procedures.

8.1 How Learning Differs from Pure Optimization

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P , that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function $J(\boldsymbol{\theta})$ in the hope that doing so will improve P . This is in contrast to pure optimization, where minimizing J is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions.

Typically, the cost function can be written as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim \hat{p}_{\text{data}}} L(f(\mathbf{x}; \boldsymbol{\theta}), y), \quad (8.1)$$

where L is the per-example loss function, $f(\mathbf{x}; \boldsymbol{\theta})$ is the predicted output when the input is \mathbf{x} , and \hat{p}_{data} is the empirical distribution. In the supervised learning case, y is the target output. Throughout this chapter, we develop the unregularized supervised case, where the arguments to L are $f(\mathbf{x}; \boldsymbol{\theta})$ and y . It is trivial to extend this development, for example, to include $\boldsymbol{\theta}$ or \mathbf{x} as arguments, or to exclude y as arguments, to develop various forms of regularization or unsupervised learning.

Equation 8.1 defines an objective function with respect to the training set. We would usually prefer to minimize the corresponding objective function where the expectation is taken across *the data-generating distribution* p_{data} rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim p_{\text{data}}} L(f(\mathbf{x}; \boldsymbol{\theta}), y). \quad (8.2)$$

8.1.1 Empirical Risk Minimization

The goal of a machine learning algorithm is to reduce the expected generalization error given by equation 8.2. This quantity is known as the **risk**. We emphasize here that the expectation is taken over the true underlying distribution p_{data} . If we knew the true distribution $p_{\text{data}}(\mathbf{x}, y)$, risk minimization would be an optimization

task solvable by an optimization algorithm. When we do not know $p_{\text{data}}(\mathbf{x}, y)$ but only have a training set of samples, however, we have a machine learning problem.

The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\mathbf{x}, y)$ with the empirical distribution $\hat{p}(\mathbf{x}, y)$ defined by the training set. We now minimize the **empirical risk**

$$\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}(\mathbf{x}, \mathbf{y})}[L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}), \quad (8.3)$$

where m is the number of training examples.

The training process based on minimizing this average training error is known as **empirical risk minimization**. In this setting, machine learning is still very similar to straightforward optimization. Rather than optimizing the risk directly, we optimize the empirical risk and hope that the risk decreases significantly as well. A variety of theoretical results establish conditions under which the true risk can be expected to decrease by various amounts.

Nonetheless, empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. In many cases, empirical risk minimization is not really feasible. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as 0-1 loss, have no useful derivatives (the derivative is either zero or undefined everywhere). These two problems mean that, in the context of deep learning, we rarely use empirical risk minimization. Instead, we must use a slightly different approach, in which the quantity that we actually optimize is even more different from the quantity that we truly want to optimize.

8.1.2 Surrogate Loss Functions and Early Stopping

Sometimes, the loss function we actually care about (say, classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier (Marcotte and Savard, 1992). In such situations, one typically optimizes a **surrogate loss function** instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss. The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input, and if the model can do that well, then it can pick the classes that yield the least classification error in expectation.

In some cases, a surrogate loss function actually results in being able to learn more. For example, the test set 0-1 loss often continues to decrease for a long time after the training set 0-1 loss has reached zero, when training using the log-likelihood surrogate. This is because even when the expected 0-1 loss is zero, one can improve the robustness of the classifier by further pushing the classes apart from each other, obtaining a more confident and reliable classifier, thus extracting more information from the training data than would have been possible by simply minimizing the average 0-1 loss on the training set.

A very important difference between optimization in general and optimization as we use it for training algorithms is that training algorithms do not usually halt at a local minimum. Instead, a machine learning algorithm usually minimizes a surrogate loss function but halts when a convergence criterion based on early stopping (section 7.8) is satisfied. Typically the early stopping criterion is based on the true underlying loss function, such as 0-1 loss measured on a validation set, and is designed to cause the algorithm to halt whenever overfitting begins to occur. Training often halts while the surrogate loss function still has large derivatives, which is very different from the pure optimization setting, where an optimization algorithm is considered to have converged when the gradient becomes very small.

8.1.3 Batch and Minibatch Algorithms

One aspect of machine learning algorithms that separates them from general optimization algorithms is that the objective function usually decomposes as a sum over the training examples. Optimization algorithms for machine learning typically compute each update to the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function.

For example, maximum likelihood estimation problems, when viewed in log space, decompose into a sum over each example:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \quad (8.4)$$

Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}, y; \boldsymbol{\theta}). \quad (8.5)$$

Most of the properties of the objective function J used by most of our optimization algorithms are also expectations over the training set. For example, the

most commonly used property is the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\mathbf{x}, y; \boldsymbol{\theta}). \quad (8.6)$$

Computing this expectation exactly is very expensive because it requires evaluating the model on every example in the entire dataset. In practice, we can compute these expectations by randomly sampling a small number of examples from the dataset, then taking the average over only those examples.

Recall that the standard error of the mean (equation 5.46) estimated from n samples is given by σ / \sqrt{n} , where σ is the true standard deviation of the value of the samples. The denominator of \sqrt{n} shows that there are less than linear returns to using more examples to estimate the gradient. Compare two hypothetical estimates of the gradient, one based on 100 examples and another based on 10,000 examples. The latter requires 100 times more computation than the former but reduces the standard error of the mean only by a factor of 10. Most optimization algorithms converge much faster (in terms of total computation, not in terms of number of updates) if they are allowed to rapidly compute approximate estimates of the gradient rather than slowly computing the exact gradient.

Another consideration motivating statistical estimation of the gradient from a small number of samples is redundancy in the training set. In the worst case, all m samples in the training set could be identical copies of each other. A sampling-based estimate of the gradient could compute the correct gradient with a single sample, using m times less computation than the naive approach. In practice, we are unlikely to encounter this worst-case situation, but we may find large numbers of examples that all make very similar contributions to the gradient.

Optimization algorithms that use the entire training set are called **batch** or **deterministic** gradient methods, because they process all the training examples simultaneously in a large batch. This terminology can be somewhat confusing because the word “batch” is also often used to describe the minibatch used by minibatch stochastic gradient descent. Typically the term “batch gradient descent” implies the use of the full training set, while the use of the term “batch” to describe a group of examples does not. For example, it is common to use the term “batch size” to describe the size of a minibatch.

Optimization algorithms that use only a single example at a time are sometimes called **stochastic** and sometimes **online** methods. The term “online” is usually reserved for when the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.

Most algorithms used for deep learning fall somewhere in between, using more

than one but fewer than all the training examples. These were traditionally called **minibatch** or **minibatch stochastic** methods, and it is now common to call them simply **stochastic** methods.

The canonical example of a stochastic method is stochastic gradient descent, presented in detail in section 8.3.1.

Minibatch sizes are generally driven by the following factors:

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect (Wilson and Martinez, 2003), perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batch size might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient. The total runtime can be very high as a result of the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.

Different kinds of algorithms use different kinds of information from the mini-batch in various ways. Some algorithms are more sensitive to sampling error than others, either because they use information that is difficult to estimate accurately with few samples, or because they use information in ways that amplify sampling errors more. Methods that compute updates based only on the gradient \mathbf{g} are usually relatively robust and can handle smaller batch sizes, like 100. Second-order methods, which also use the Hessian matrix \mathbf{H} and compute updates such as $\mathbf{H}^{-1}\mathbf{g}$, typically require much larger batch sizes, like 10,000. These large batch sizes are required to minimize fluctuations in the estimates of $\mathbf{H}^{-1}\mathbf{g}$. Suppose

that \mathbf{H} is estimated perfectly but has a poor condition number. Multiplication by \mathbf{H} or its inverse amplifies pre-existing errors, in this case, estimation errors in \mathbf{g} . Very small changes in the estimate of \mathbf{g} can thus cause large changes in the update $\mathbf{H}^{-1}\mathbf{g}$, even if \mathbf{H} is estimated perfectly. Of course, \mathbf{H} is estimated only approximately, so the update $\mathbf{H}^{-1}\mathbf{g}$ will contain even more error than we would predict from applying a poorly conditioned operation to the estimate of \mathbf{g} .

It is also crucial that the minibatches be selected randomly. Computing an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent. We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent minibatches of examples should also be independent from each other. Many datasets are most naturally arranged in a way where successive examples are highly correlated. For example, we might have a dataset of medical data with a long list of blood sample test results. This list might be arranged so that first we have five blood samples taken at different times from the first patient, then we have three blood samples taken from the second patient, then the blood samples from the third patient, and so on. If we were to draw examples in order from this list, then each of our minibatches would be extremely biased, because it would represent primarily one patient out of the many patients in the dataset. In cases such as these, where the order of the dataset holds some significance, it is necessary to shuffle the examples before selecting minibatches. For very large datasets, for example, datasets containing billions of examples in a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch. Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion. This will impose a fixed set of possible minibatches of consecutive examples that all models trained thereafter will use, and each individual model will be forced to reuse this ordering every time it passes through the training data. This deviation from true random selection does not seem to have a significant detrimental effect. Failing to ever shuffle the examples in any way can seriously reduce the effectiveness of the algorithm.

Many optimization problems in machine learning decompose over examples well enough that we can compute entire separate updates over different examples in parallel. In other words, we can compute the update that minimizes $J(\mathbf{X})$ for one minibatch of examples \mathbf{X} at the same time that we compute the update for several other minibatches. Such asynchronous parallel distributed approaches are discussed further in section 12.1.3.

An interesting motivation for minibatch stochastic gradient descent is that it follows the gradient of the true *generalization error* (equation 8.2) as long as no

examples are repeated. Most implementations of minibatch stochastic gradient descent shuffle the dataset once and then pass through it multiple times. On the first pass, each minibatch is used to compute an unbiased estimate of the true generalization error. On the second pass, the estimate becomes biased because it is formed by resampling values that have already been used, rather than obtaining new fair samples from the data-generating distribution.

The fact that stochastic gradient descent minimizes generalization error is easiest to see in online learning, where examples or minibatches are drawn from a **stream** of data. In other words, instead of receiving a fixed-size training set, the learner is similar to a living being who sees a new example at each instant, with every example (\mathbf{x}, y) coming from the data-generating distribution $p_{\text{data}}(\mathbf{x}, y)$. In this scenario, examples are never repeated; every experience is a fair sample from p_{data} .

The equivalence is easiest to derive when both \mathbf{x} and y are discrete. In this case, the generalization error (equation 8.2) can be written as a sum

$$J^*(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \sum_y p_{\text{data}}(\mathbf{x}, y) L(f(\mathbf{x}; \boldsymbol{\theta}), y), \quad (8.7)$$

with the exact gradient

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \sum_y p_{\text{data}}(\mathbf{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), y). \quad (8.8)$$

We have already seen the same fact demonstrated for the log-likelihood in equation 8.5 and equation 8.6; we observe now that this holds for other functions L besides the likelihood. A similar result can be derived when \mathbf{x} and y are continuous, under mild assumptions regarding p_{data} and L .

Hence, we can obtain an unbiased estimator of the exact gradient of the generalization error by sampling a minibatch of examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $y^{(i)}$ from the data-generating distribution p_{data} , then computing the gradient of the loss with respect to the parameters for that minibatch:

$$\hat{\mathbf{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (8.9)$$

Updating $\boldsymbol{\theta}$ in the direction of $\hat{\mathbf{g}}$ performs SGD on the generalization error.

Of course, this interpretation applies only when examples are not reused. Nonetheless, it is usually best to make several passes through the training set, unless the training set is extremely large. When multiple such epochs are used,

only the first epoch follows the unbiased gradient of the generalization error, but of course, the additional epochs usually provide enough benefit due to decreased training error to offset the harm they cause by increasing the gap between training error and test error.

With some datasets growing rapidly in size, faster than computing power, it is becoming more common for machine learning applications to use each training example only once or even to make an incomplete pass through the training set. When using an extremely large training set, overfitting is not an issue, so underfitting and computational efficiency become the predominant concerns. See also [Bottou and Bousquet \(2008\)](#) for a discussion of the effect of computational bottlenecks on generalization error, as the number of training examples grows.

8.2 Challenges in Neural Network Optimization

Optimization in general is an extremely difficult task. Traditionally, machine learning has avoided the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex. When training neural networks, we must confront the general nonconvex case. Even convex optimization is not without its complications. In this section, we summarize several of the most prominent challenges involved in optimization for training deep models.

8.2.1 Ill-Conditioning

Some challenges arise even when optimizing convex functions. Of these, the most prominent is ill-conditioning of the Hessian matrix \mathbf{H} . This is a very general problem in most numerical optimization, convex or otherwise, and is described in more detail in section [4.3.1](#).

The ill-conditioning problem is generally believed to be present in neural network training problems. Ill-conditioning can manifest by causing SGD to get “stuck” in the sense that even very small steps increase the cost function.

Recall from equation [4.9](#) that a second-order Taylor series expansion of the cost function predicts that a gradient descent step of $-\epsilon\mathbf{g}$ will add

$$\frac{1}{2}\epsilon^2\mathbf{g}^\top \mathbf{H}\mathbf{g} - \epsilon\mathbf{g}^\top \mathbf{g} \tag{8.10}$$

to the cost. Ill-conditioning of the gradient becomes a problem when $\frac{1}{2}\epsilon^2\mathbf{g}^\top \mathbf{H}\mathbf{g}$ exceeds $\epsilon\mathbf{g}^\top \mathbf{g}$. To determine whether ill-conditioning is detrimental to a neural

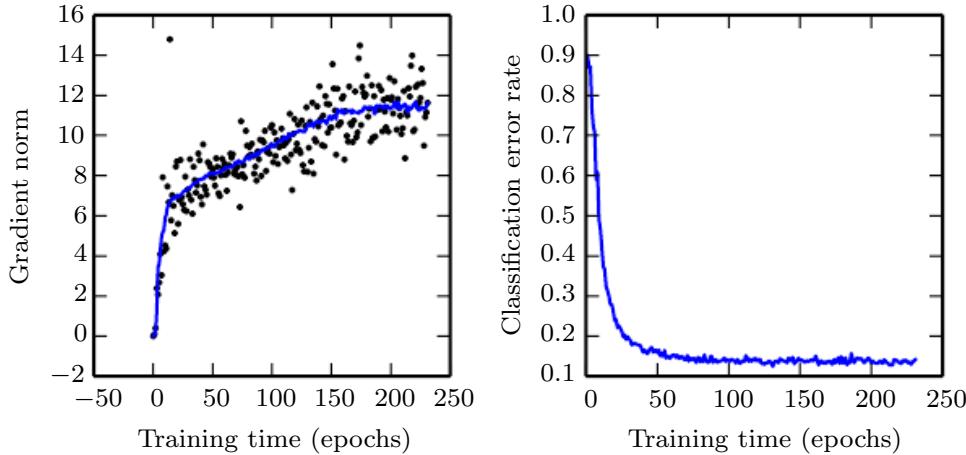


Figure 8.1: Gradient descent often does not arrive at a critical point of any kind. In this example, the gradient norm increases throughout training of a convolutional network used for object detection. (*Left*)A scatterplot showing how the norms of individual gradient evaluations are distributed over time. To improve legibility, only one gradient norm is plotted per epoch. The running average of all gradient norms is plotted as a solid curve. The gradient norm clearly increases over time, rather than decreasing as we would expect if the training process converged to a critical point. (*Right*)Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.

network training task, one can monitor the squared gradient norm $\mathbf{g}^\top \mathbf{g}$ and the $\mathbf{g}^\top \mathbf{H}\mathbf{g}$ term. In many cases, the gradient norm does not shrink significantly throughout learning, but the $\mathbf{g}^\top \mathbf{H}\mathbf{g}$ term grows by more than an order of magnitude. The result is that learning becomes very slow despite the presence of a strong gradient because the learning rate must be shrunk to compensate for even stronger curvature. Figure 8.1 shows an example of the gradient increasing significantly during the successful training of a neural network.

Though ill-conditioning is present in other settings besides neural network training, some of the techniques used to combat it in other contexts are less applicable to neural networks. For example, Newton's method is an excellent tool for minimizing convex functions with poorly conditioned Hessian matrices, but as we argue in subsequent sections, Newton's method requires significant modification before it can be applied to neural networks.

8.2.2 Local Minima

One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is guaranteed to be a global minimum. Some convex functions have a flat region at the bottom rather than a single global minimum point, but any point within such a flat region is an acceptable solution. When optimizing a convex function, we know that we have reached a good solution if we find a critical point of any kind.

With nonconvex functions, such as neural nets, it is possible to have many local minima. Indeed, nearly any deep model is essentially guaranteed to have an extremely large number of local minima. As we will see, however, this is not necessarily a major problem.

Neural networks and any models with multiple equivalently parametrized latent variables all have multiple local minima because of the **model identifiability** problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model’s parameters. Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other. For example, we could take a neural network and modify layer 1 by swapping the incoming weight vector for unit i with the incoming weight vector for unit j , then do the same for the outgoing weight vectors. If we have m layers with n units each, then there are $n!^m$ ways of arranging the hidden units. This kind of nonidentifiability is known as **weight space symmetry**.

In addition to weight space symmetry, many kinds of neural networks have additional causes of nonidentifiability. For example, in any rectified linear or maxout network, we can scale all the incoming weights and biases of a unit by α if we also scale all its outgoing weights by $\frac{1}{\alpha}$. This means that—if the cost function does not include terms such as weight decay that depend directly on the weights rather than the models’ outputs—every local minimum of a rectified linear or maxout network lies on an $(m \times n)$ -dimensional hyperbola of equivalent local minima.

These model identifiability issues mean that a neural network cost function can have an extremely large or even uncountably infinite amount of local minima. However, all these local minima arising from nonidentifiability are equivalent to each other in cost function value. As a result, these local minima are not a problematic form of nonconvexity.

Local minima can be problematic if they have high cost in comparison to the global minimum. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minimum ([Sontag](#)

and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992). If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

Whether networks of practical interest have many local minima of high cost and whether optimization algorithms encounter them remain open questions. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case. The problem remains an active area of research, but experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost (Saxe *et al.*, 2013; Dauphin *et al.*, 2014; Goodfellow *et al.*, 2015; Choromanska *et al.*, 2014).

Many practitioners attribute nearly all difficulty with neural network optimization to local minima. We encourage practitioners to carefully test for specific problems. A test that can rule out local minima as the problem is plotting the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is neither local minima nor any other kind of critical point. In high-dimensional spaces, positively establishing that local minima are the problem can be very difficult. Many structures other than local minima also have small gradients.

8.2.3 Plateaus, Saddle Points and Other Flat Regions

For many high-dimensional nonconvex functions, local minima (and maxima) are in fact rare compared to another kind of point with zero gradient: a saddle point. Some points around a saddle point have greater cost than the saddle point, while others have a lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues. Points lying along eigenvectors associated with positive eigenvalues have greater cost than the saddle point, while points lying along negative eigenvalues have lower value. We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section. See figure 4.5 for an illustration.

Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher-dimensional spaces, local minima are rare, and saddle points are more common. For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ of this type, the expected ratio of the number of saddle points to local minima grows exponentially with n . To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The

Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In a single dimension, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n -dimensional space, it is exponentially unlikely that all n coin tosses will be heads. See [Dauphin et al. \(2014\)](#) for a review of the relevant theoretical work.

An amazing property of many random functions is that the eigenvalues of the Hessian become more likely to be positive as we reach regions of lower cost. In our coin tossing analogy, this means we are more likely to have our coin come up heads n times if we are at a critical point with low cost. It also means that local minima are much more likely to have low cost than high cost. Critical points with high cost are far more likely to be saddle points. Critical points with extremely high cost are more likely to be local maxima.

This happens for many classes of random functions. Does it happen for neural networks? [Baldi and Hornik \(1989\)](#) showed theoretically that shallow autoencoders (feedforward networks trained to copy their input to their output, described in chapter 14) with no nonlinearities have global minima and saddle points but no local minima with higher cost than the global minimum. They observed without proof that these results extend to deeper networks without nonlinearities. The output of such networks is a linear function of their input, but they are useful to study as a model of nonlinear neural networks because their loss function is a nonconvex function of their parameters. Such networks are essentially just multiple matrices composed together. [Saxe et al. \(2013\)](#) provided exact solutions to the complete learning dynamics in such networks and showed that learning in these models captures many of the qualitative features observed in the training of deep models with nonlinear activation functions. [Dauphin et al. \(2014\)](#) showed experimentally that real neural networks also have loss functions that contain very many high-cost saddle points. [Choromanska et al. \(2014\)](#) provided additional theoretical arguments, showing that another class of high-dimensional random functions related to neural networks does so as well.

What are the implications of the proliferation of saddle points for training algorithms? For first-order optimization, algorithms that use only gradient information, the situation is unclear. The gradient can often become very small near a saddle point. On the other hand, gradient descent empirically seems able to escape saddle points in many cases. [Goodfellow et al. \(2015\)](#) provided visualizations of several learning trajectories of state-of-the-art neural networks, with an example given in figure 8.2. These visualizations show a flattening of the cost function near a prominent saddle point, where the weights are all zero, but they also show the gradient descent trajectory rapidly escaping this region. [Goodfellow et al. \(2015\)](#)

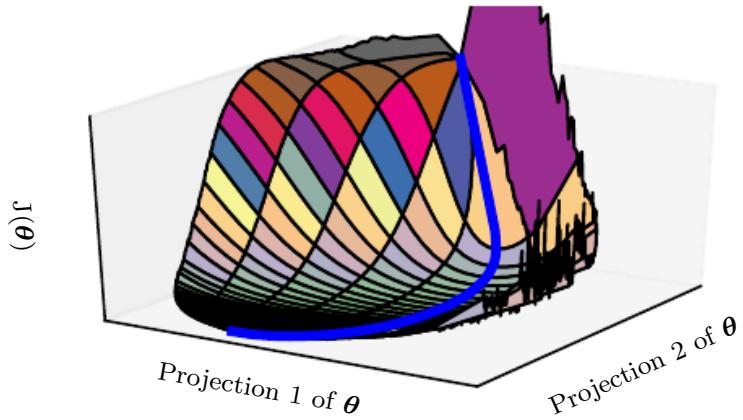


Figure 8.2: A visualization of the cost function of a neural network. These visualizations appear similar for feedforward neural networks, convolutional networks, and recurrent networks applied to real object recognition and natural language processing tasks. Surprisingly, these visualizations usually do not show many conspicuous obstacles. Prior to the success of stochastic gradient descent for training very large models beginning in roughly 2012, neural net cost function surfaces were generally believed to have much more nonconvex structure than is revealed by these projections. The primary obstacle revealed by this projection is a saddle point of high cost near where the parameters are initialized, but, as indicated by the blue path, the SGD training trajectory escapes this saddle point readily. Most of training time is spent traversing the relatively flat valley of the cost function, perhaps because of high noise in the gradient, poor conditioning of the Hessian matrix in this region, or simply the need to circumnavigate the tall “mountain” visible in the figure via an indirect arcing path. Image adapted with permission from [Goodfellow et al. \(2015\)](#).

also argue that continuous-time gradient descent may be shown analytically to be repelled from, rather than attracted to, a nearby saddle point, but the situation may be different for more realistic uses of gradient descent.

For Newton’s method, saddle points clearly constitute a problem. Gradient descent is designed to move “downhill” and is not explicitly designed to seek a critical point. Newton’s method, however, is designed to solve for a point where the gradient is zero. Without appropriate modification, it can jump to a saddle point. The proliferation of saddle points in high-dimensional spaces presumably explains why second-order methods have not succeeded in replacing gradient descent for neural network training. [Dauphin et al. \(2014\)](#) introduced a **saddle-free Newton method** for second-order optimization and showed that it improves significantly over the traditional version. Second-order methods remain difficult to scale to large neural networks, but this saddle-free approach holds promise if it can be scaled.

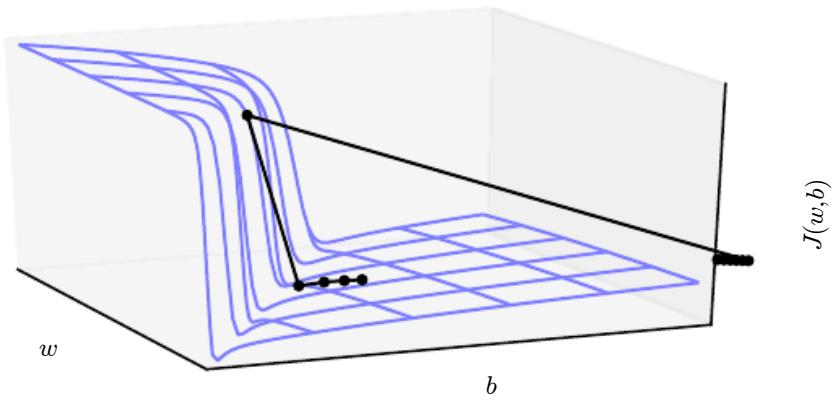


Figure 8.3: The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that has been done. Figure adapted with permission from [Pascanu et al. \(2013\)](#).

There are other kinds of points with zero gradient besides minima and saddle points. Maxima are much like saddle points from the perspective of optimization—many algorithms are not attracted to them, but unmodified Newton’s method is. Maxima of many classes of random functions become exponentially rare in high-dimensional space, just as minima do.

There may also be wide, flat regions of constant value. In these locations, the gradient and the Hessian are all zero. Such degenerate locations pose major problems for all numerical optimization algorithms. In a convex problem, a wide, flat region must consist entirely of global minima, but in a general optimization problem, such a region could correspond to a high value of the objective function.

8.2.4 Cliffs and Exploding Gradients

Neural networks with many layers often have extremely steep regions resembling cliffs, as illustrated in figure 8.3. These result from the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off the cliff structure altogether.

The cliff can be dangerous whether we approach it from above or from below,

but fortunately its most serious consequences can be avoided using the **gradient clipping** heuristic described in section 10.11.1. The basic idea is to recall that the gradient specifies not the optimal step size, but only the optimal direction within an infinitesimal region. When the traditional gradient descent algorithm proposes making a very large step, the gradient clipping heuristic intervenes to reduce the step size, making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent. Cliff structures are most common in the cost functions for recurrent neural networks, because such models involve a multiplication of many factors, with one factor for each time step. Long temporal sequences thus incur an extreme amount of multiplication.

8.2.5 Long-Term Dependencies

Another difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep. Feedforward networks with many layers have such deep computational graphs. So do recurrent networks, described in chapter 10, which construct very deep computational graphs by repeatedly applying the same operation at each time step of a long temporal sequence. Repeated application of the same parameters gives rise to especially pronounced difficulties.

For example, suppose that a computational graph contains a path that consists of repeatedly multiplying by a matrix \mathbf{W} . After t steps, this is equivalent to multiplying by \mathbf{W}^t . Suppose that \mathbf{W} has an eigendecomposition $\mathbf{W} = \mathbf{V}\text{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1}$. In this simple case, it is straightforward to see that

$$\mathbf{W}^t = (\mathbf{V}\text{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1})^t = \mathbf{V}\text{diag}(\boldsymbol{\lambda})^t\mathbf{V}^{-1}. \quad (8.11)$$

Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude. The **vanishing and exploding gradient problem** refers to the fact that gradients through such a graph are also scaled according to $\text{diag}(\boldsymbol{\lambda})^t$. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.

The repeated multiplication by \mathbf{W} at each time step described here is very similar to the **power method** algorithm used to find the largest eigenvalue of a matrix \mathbf{W} and the corresponding eigenvector. From this point of view it is not surprising that $\mathbf{x}^\top \mathbf{W}^t$ will eventually discard all components of \mathbf{x} that are orthogonal to the principal eigenvector of \mathbf{W} .

Recurrent networks use the same matrix \mathbf{W} at each time step, but feedforward networks do not, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem ([Sussillo, 2014](#)).

We defer further discussion of the challenges of training recurrent networks until section [10.7](#), after recurrent networks have been described in more detail.

8.2.6 Inexact Gradients

Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix. In practice, we usually have only a noisy or even biased estimate of these quantities. Nearly every deep learning algorithm relies on sampling-based estimates, at least insofar as using a minibatch of training examples to compute the gradient.

In other cases, the objective function we want to minimize is actually intractable. When the objective function is intractable, typically its gradient is intractable as well. In such cases we can only approximate the gradient. These issues mostly arise with the more advanced models we cover in part [III](#). For example, contrastive divergence gives a technique for approximating the gradient of the intractable log-likelihood of a Boltzmann machine.

Various neural network optimization algorithms are designed to account for imperfections in the gradient estimate. One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss.

8.2.7 Poor Correspondence between Local and Global Structure

Many of the problems we have discussed so far correspond to properties of the loss function at a single point—it can be difficult to make a single step if $J(\boldsymbol{\theta})$ is poorly conditioned at the current point $\boldsymbol{\theta}$, or if $\boldsymbol{\theta}$ lies on a cliff, or if $\boldsymbol{\theta}$ is a saddle point hiding the opportunity to make progress downhill from the gradient.

It is possible to overcome all these problems at a single point and still perform poorly if the direction that results in the most improvement locally does not point toward distant regions of much lower cost.

[Goodfellow et al. \(2015\)](#) argue that much of the runtime of training is due to the length of the trajectory needed to arrive at the solution. Figure [8.2](#) shows that the learning trajectory spends most of its time tracing out a wide arc around a mountain-shaped structure.

Much of research into the difficulties of optimization has focused on whether

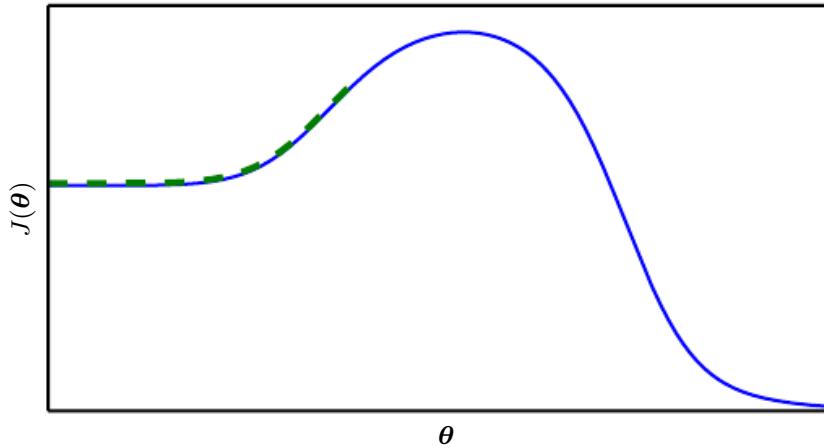


Figure 8.4: Optimization based on local downhill moves can fail if the local surface does not point toward the global solution. Here we provide an example of how this can occur, even if there are no saddle points or local minima. This example cost function contains only asymptotes toward low values, not minima. The main cause of difficulty in this case is being initialized on the wrong side of the “mountain” and not being able to traverse it. In higher-dimensional space, learning algorithms can often circumnavigate such mountains, but the trajectory associated with doing so may be long and result in excessive training time, as illustrated in figure 8.2.

training arrives at a global minimum, a local minimum, or a saddle point, but in practice, neural networks do not arrive at a critical point of any kind. Figure 8.1 shows that neural networks often do not arrive at a region of small gradient. Indeed, such critical points do not even necessarily exist. For example, the loss function $-\log p(y | \mathbf{x}; \boldsymbol{\theta})$ can lack a global minimum point and instead asymptotically approach some value as the model becomes more confident. For a classifier with discrete y and $p(y | \mathbf{x})$ provided by a softmax, the negative log-likelihood can become arbitrarily close to zero if the model is able to correctly classify every example in the training set, but it is impossible to actually reach the value of zero. Likewise, a model of real values $p(y | \mathbf{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ can have negative log-likelihood that asymptotes to negative infinity—if $f(\boldsymbol{\theta})$ is able to correctly predict the value of all training set y targets, the learning algorithm will increase β without bound. See figure 8.4 for an example of a failure of local optimization to find a good cost function value even in the absence of any local minima or saddle points.

Future research will need to develop further understanding of the factors that influence the length of the learning trajectory and better characterize the outcome of the process.

Many existing research directions are aimed at finding good initial points for problems that have difficult global structure, rather than at developing algorithms that use nonlocal moves.

Gradient descent and essentially all learning algorithms that are effective for training neural networks are based on making small local moves. The previous sections have primarily focused on how the correct direction of these local moves can be difficult to compute. We may be able to compute some properties of the objective function, such as its gradient, only approximately, with bias or variance in our estimate of the correct direction. In these cases, local descent may or may not define a reasonably short path to a valid solution, but we are not actually able to follow the local descent path. The objective function may have issues, such as poor conditioning or discontinuous gradients, causing the region where the gradient provides a good model of the objective function to be very small. In these cases, local descent with steps of size ϵ may define a reasonably short path to the solution, but we are only able to compute the local descent direction with steps of size $\delta \ll \epsilon$. In these cases, local descent may define a path to the solution, but the path contains many steps, so following it incurs a high computational cost. Sometimes local information provides us no guide, such as when the function has a wide flat region, or if we manage to land exactly on a critical point (usually this latter scenario only happens to methods that solve explicitly for critical points, such as Newton's method). In these cases, local descent does not define a path to a solution at all. In other cases, local moves can be too greedy and lead us along a path that moves downhill but away from any solution, as in figure 8.4, or along an unnecessarily long trajectory to the solution, as in figure 8.2. Currently, we do not understand which of these problems are most relevant to making neural network optimization difficult, and this is an active area of research.

Regardless of which of these problems are most significant, all of them might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if we are able to initialize learning within that well-behaved region. This last view suggests research into choosing good initial points for traditional optimization algorithms to use.

8.2.8 Theoretical Limits of Optimization

Several theoretical results show that there are limits on the performance of any optimization algorithm we might design for neural networks (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997). Typically these results have little bearing on the use of neural networks in practice.

Some theoretical results apply only when the units of a neural network output discrete values. Most neural network units output smoothly increasing values that make optimization via local search feasible. Some theoretical results show that there exist problem classes that are intractable, but it can be difficult to tell whether a particular problem falls into that class. Other results show that finding a solution for a network of a given size is intractable, but in practice we can find a solution easily by using a larger network for which many more parameter settings correspond to an acceptable solution. Moreover, in the context of neural network training, we usually do not care about finding the exact minimum of a function, but seek only to reduce its value sufficiently to obtain good generalization error. Theoretical analysis of whether an optimization algorithm can accomplish this goal is extremely difficult. Developing more realistic bounds on the performance of optimization algorithms therefore remains an important goal for machine learning research.

8.3 Basic Algorithms

We have previously introduced the gradient descent (section 4.3) algorithm that follows the gradient of an entire training set downhill. This may be accelerated considerably by using stochastic gradient descent to follow the gradient of randomly selected minibatches downhill, as discussed in section 5.9 and section 8.1.3.

8.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular. As discussed in section 8.1.3, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data-generating distribution.

Algorithm 8.1 shows how to follow this estimate of the gradient downhill.

A crucial parameter for the SGD algorithm is the learning rate. Previously, we have described SGD as using a fixed learning rate ϵ . In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration k as ϵ_k .

This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. By comparison, the true gradient of the total cost function becomes small and then $\mathbf{0}$ when we approach and reach a minimum using batch gradient

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \dots$

Require: Initial parameter θ

$k \leftarrow 1$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Apply update: $\theta \leftarrow \theta - \epsilon_k \hat{\mathbf{g}}$

$k \leftarrow k + 1$

end while

descent, so batch gradient descent can use a fixed learning rate. A sufficient condition to guarantee convergence of SGD is that

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \tag{8.12}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \tag{8.13}$$

In practice, it is common to decay the learning rate linearly until iteration τ :

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_{\tau} \tag{8.14}$$

with $\alpha = \frac{k}{\tau}$. After iteration τ , it is common to leave ϵ constant.

The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time. This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism. When using the linear schedule, the parameters to choose are ϵ_0 , ϵ_{τ} , and τ . Usually τ may be set to the number of iterations required to make a few hundred passes through the training set. Usually ϵ_{τ} should be set to roughly 1 percent the value of ϵ_0 . The main question is how to set ϵ_0 . If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function, such as the cost function arising from the use of dropout. If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value. Typically, the optimal initial learning rate, in terms of total training time and the

final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability.

The most important property of SGD and related minibatch or online gradient-based optimization is that computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large. For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set.

To study the convergence rate of an optimization algorithm it is common to measure the **excess error** $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the amount by which the current cost function exceeds the minimum possible cost. When SGD is applied to a convex problem, the excess error is $O(\frac{1}{\sqrt{k}})$ after k iterations, while in the strongly convex case, it is $O(\frac{1}{k})$. These bounds cannot be improved unless extra conditions are assumed. Batch gradient descent enjoys better convergence rates than stochastic gradient descent in theory. However, the Cramér-Rao bound (Cramér, 1946; Rao, 1945) states that generalization error cannot decrease faster than $O(\frac{1}{k})$. Bottou and Bousquet (2008) argue that it therefore may not be worthwhile to pursue an optimization algorithm that converges faster than $O(\frac{1}{k})$ for machine learning tasks—faster convergence presumably corresponds to overfitting. Moreover, the asymptotic analysis obscures many advantages that stochastic gradient descent has after a small number of steps. With large datasets, the ability of SGD to make rapid initial progress while evaluating the gradient for very few examples outweighs its slow asymptotic convergence. Most of the algorithms described in the remainder of this chapter achieve benefits that matter in practice but are lost in the constant factors obscured by the $O(\frac{1}{k})$ asymptotic analysis. One can also trade off the benefits of both batch and stochastic gradient descent by gradually increasing the minibatch size during the course of learning.

For more information on SGD, see Bottou (1998).

8.3.2 Momentum

While stochastic gradient descent remains a popular optimization strategy, learning with it can sometimes be slow. The method of momentum (Polyak, 1964) is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move

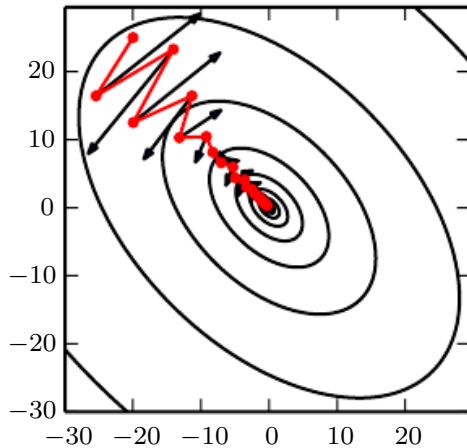


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

in their direction. The effect of momentum is illustrated in figure 8.5.

Formally, the momentum algorithm introduces a variable \mathbf{v} that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name **momentum** derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton’s laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector \mathbf{v} may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right), \quad (8.15)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}. \quad (8.16)$$

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ , momentum parameter α

Require: Initial parameter θ , initial velocity v

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

 Compute velocity update: $v \leftarrow \alpha v - \epsilon \mathbf{g}$.

 Apply update: $\theta \leftarrow \theta + v$.

end while

The velocity v accumulates the gradient elements $\nabla_{\theta} (\frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}))$. The larger α is relative to ϵ , the more previous gradients affect the current direction. The SGD algorithm with momentum is given in algorithm 8.2.

Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a *sequence* of gradients are. The step size is largest when many successive gradients point in exactly the same direction. If the momentum algorithm always observes gradient \mathbf{g} , then it will accelerate in the direction of $-\mathbf{g}$, until reaching a terminal velocity where the size of each step is

$$\frac{\epsilon \|\mathbf{g}\|}{1 - \alpha}. \quad (8.17)$$

It is thus helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\alpha}$. For example, $\alpha = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

Common values of α used in practice include 0.5, 0.9, and 0.99. Like the learning rate, α may also be adapted over time. Typically it begins with a small value and is later raised. Adapting α over time is less important than shrinking ϵ over time.

We can view the momentum algorithm as simulating a particle subject to continuous-time Newtonian dynamics. The physical analogy can help build intuition for how the momentum and gradient descent algorithms behave.

The position of the particle at any point in time is given by $\theta(t)$. The particle experiences net force $f(t)$. This force causes the particle to accelerate:

$$f(t) = \frac{\partial^2}{\partial t^2} \theta(t). \quad (8.18)$$

Rather than viewing this as a second-order differential equation of the position, we can introduce the variable $\mathbf{v}(t)$ representing the velocity of the particle at time t and rewrite the Newtonian dynamics as a first-order differential equation:

$$\mathbf{v}(t) = \frac{\partial}{\partial t} \boldsymbol{\theta}(t), \quad (8.19)$$

$$\mathbf{f}(t) = \frac{\partial}{\partial t} \mathbf{v}(t). \quad (8.20)$$

The momentum algorithm then consists of solving the differential equations via numerical simulation. A simple numerical method for solving differential equations is Euler's method, which simply consists of simulating the dynamics defined by the equation by taking small, finite steps in the direction of each gradient.

This explains the basic form of the momentum update, but what specifically are the forces? One force is proportional to the negative gradient of the cost function: $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This force pushes the particle downhill along the cost function surface. The gradient descent algorithm would simply take a single step based on each gradient, but the Newtonian scenario used by the momentum algorithm instead uses this force to alter the velocity of the particle. We can think of the particle as being like a hockey puck sliding down an icy surface. Whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it begins to go uphill again.

One other force is necessary. If the only force is the gradient of the cost function, then the particle might never come to rest. Imagine a hockey puck sliding down one side of a valley and straight up the other side, oscillating back and forth forever, assuming the ice is perfectly frictionless. To resolve this problem, we add one other force, proportional to $-\mathbf{v}(t)$. In physics terminology, this force corresponds to viscous drag, as if the particle must push through a resistant medium such as syrup. This causes the particle to gradually lose energy over time and eventually converge to a local minimum.

Why do we use $-\mathbf{v}(t)$ and viscous drag in particular? Part of the reason to use $-\mathbf{v}(t)$ is mathematical convenience—an integer power of the velocity is easy to work with. Yet other physical systems have other kinds of drag based on other integer powers of the velocity. For example, a particle traveling through the air experiences turbulent drag, with force proportional to the square of the velocity, while a particle moving along the ground experiences dry friction, with a force of constant magnitude. We can reject each of these options. Turbulent drag, proportional to the square of the velocity, becomes very weak when the velocity is small. It is not powerful enough to force the particle to come to rest. A particle with a nonzero initial velocity that experiences only the force of turbulent drag

will move away from its initial position forever, with the distance from the starting point growing like $O(\log t)$. We must therefore use a lower power of the velocity. If we use a power of zero, representing dry friction, then the force is too strong. When the force due to the gradient of the cost function is small but nonzero, the constant force due to friction can cause the particle to come to rest before reaching a local minimum. Viscous drag avoids both of these problems—it is weak enough that the gradient can continue to cause motion until a minimum is reached, but strong enough to prevent motion if the gradient does not justify moving.

8.3.3 Nesterov Momentum

Sutskever *et al.* (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov’s accelerated gradient method (Nesterov, 1983, 2004). The update rules in this case are given by

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^m L\left(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta} + \alpha \mathbf{v}), \mathbf{y}^{(i)}\right) \right], \quad (8.21)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}, \quad (8.22)$$

where the parameters α and ϵ play a similar role as in the standard momentum method. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum, the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a *correction factor* to the standard method of momentum. The complete Nesterov momentum algorithm is presented in algorithm 8.3.

In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(1/k)$ (after k steps) to $O(1/k^2)$ as shown by Nesterov (1983). Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

8.4 Parameter Initialization Strategies

Some optimization algorithms are not iterative by nature and simply solve for a solution point. Other optimization algorithms are iterative by nature but, when applied to the right class of optimization problems, converge to acceptable solutions in an acceptable amount of time regardless of initialization. Deep learning training algorithms usually do not have either of these luxuries. Training algorithms for

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ , momentum parameter α **Require:** Initial parameter θ , initial velocity v **while** stopping criterion not met **do** Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding labels $\mathbf{y}^{(i)}$. Apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$. Compute gradient (at interim point): $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$. Compute velocity update: $v \leftarrow \alpha v - \epsilon g$. Apply update: $\theta \leftarrow \theta + v$.**end while**

deep learning models are usually iterative and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood. Most initialization strategies are based on achieving some nice properties when the network is initialized. However, we do not have a good understanding of which of these properties are preserved under which circumstances after learning begins to proceed. A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization. Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point.

Perhaps the only property known with complete certainty is that the initial parameters need to “break symmetry” between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way. Even if the

model or training algorithm is capable of using stochasticity to compute different updates for different units (for example, if one trains with dropout), it is usually best to initialize each unit to compute a different function from all the other units. This may help to make sure that no input patterns are lost in the null space of forward propagation and that no gradient patterns are lost in the null space of back-propagation. The goal of having each unit compute a different function motivates random initialization of the parameters. We could explicitly search for a large set of basis functions that are all mutually different from each other, but this often incurs a noticeable computational cost. For example, if we have at most as many outputs as inputs, we could use Gram-Schmidt orthogonalization on an initial weight matrix and be guaranteed that each unit would compute a very different function from each other unit. Random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper and unlikely to assign any units to compute the same function as each other.

Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly. Extra parameters—for example, parameters encoding the conditional variance of a prediction—are usually set to heuristically chosen constants much like the biases are.

We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter much but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry-breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or back-propagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Initial weights that are too large may, however, result in exploding values during forward propagation or back-propagation. In recurrent networks, large weights can also result in **chaos** (such extreme sensitivity to small perturbations of the input that the behavior of the deterministic forward propagation procedure appears random). To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step). Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units. These competing factors determine the ideal initial scale of the weights.

The perspectives of regularization and optimization can give very different

insights into how we should initialize a network. The optimization perspective suggests that the weights should be large enough to propagate information successfully, but some regularization concerns encourage making them smaller. The use of an optimization algorithm, such as stochastic gradient descent, that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters (whether due to getting stuck in a region of low gradient, or due to triggering some early stopping criterion based on overfitting) expresses a prior that the final parameters should be close to the initial parameters. Recall from section 7.8 that gradient descent with early stopping is equivalent to weight decay for some models. In the general case, gradient descent with early stopping is not the same as weight decay, but it does provide a loose analogy for thinking about the effect of initialization. We can think of initializing the parameters θ to θ_0 as being similar to imposing a Gaussian prior $p(\theta)$ with mean θ_0 . From this point of view, it makes sense to choose θ_0 to be near 0. This prior says that it is more likely that units do not interact with each other than that they do interact. Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. On the other hand, if we initialize θ_0 to large values, then our prior specifies which units should interact with each other, and how they should interact.

Some heuristics are available for choosing the initial scale of the weights. One heuristic is to initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$, while Glorot and Bengio (2010) suggest using the **normalized initialization**

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \quad (8.23)$$

This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance. The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no nonlinearities. Real neural networks obviously violate this assumption, but many strategies designed for the linear model perform reasonably well on its nonlinear counterparts.

Saxe *et al.* (2013) recommend initializing to random orthogonal matrices, with a carefully chosen scaling or **gain** factor g that accounts for the nonlinearity applied at each layer. They derive specific values of the scaling factor for different types of nonlinear activation functions. This initialization scheme is also motivated by a model of a deep network as a sequence of matrix multiplies without nonlinearities.

Under such a model, this initialization scheme guarantees that the total number of training iterations required to reach convergence is independent of depth.

Increasing the scaling factor g pushes the network toward the regime where activations increase in norm as they propagate forward through the network and gradients increase in norm as they propagate backward. [Sussillo \(2014\)](#) showed that setting the gain factor correctly is sufficient to train networks as deep as 1,000 layers, without needing to use orthogonal initializations. A key insight of this approach is that in feedforward networks, activations and gradients can grow or shrink on each step of forward or back-propagation, following a random walk behavior. This is because feedforward networks use a different weight matrix at each layer. If this random walk is tuned to preserve norms, then feedforward networks can mostly avoid the vanishing and exploding gradients problem that arises when the same weight matrix is used at each step, as described in section [8.2.5](#).

Unfortunately, these optimal criteria for initial weights often do not lead to optimal performance. This may be for three different reasons. First, we may be using the wrong criteria—it may not actually be beneficial to preserve the norm of a signal throughout the entire network. Second, the properties imposed at initialization may not persist after learning has begun to proceed. Third, the criteria might succeed at improving the speed of optimization but inadvertently increase generalization error. In practice, we usually need to treat the scale of the weights as a hyperparameter whose optimal value lies somewhere roughly near but not exactly equal to the theoretical predictions.

One drawback to scaling rules that set all the initial weights to have the same standard deviation, such as $\frac{1}{\sqrt{m}}$, is that every individual weight becomes extremely small when the layers become large. [Martens \(2010\)](#) introduced an alternative initialization scheme called **sparse initialization**, in which each unit is initialized to have exactly k nonzero weights. The idea is to keep the total amount of input to the unit independent from the number of inputs m without making the magnitude of individual weight elements shrink with m . Sparse initialization helps to achieve more diversity among the units at initialization time. However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values. Because it takes a long time for gradient descent to shrink “incorrect” large values, this initialization scheme can cause problems for units, such as maxout units, that have several filters that must be carefully coordinated with each other.

When computational resources allow it, it is usually a good idea to treat the initial scale of the weights for each layer as a hyperparameter, and to choose these scales using a hyperparameter search algorithm described in section [11.4.2](#), such as random search. The choice of whether to use dense or sparse initialization

can also be made a hyperparameter. Alternately, one can manually search for the best initial scales. A good rule of thumb for choosing the initial scales is to look at the range or standard deviation of activations or gradients on a single minibatch of data. If the weights are too small, the range of activations across the minibatch will shrink as the activations propagate forward through the network. By repeatedly identifying the first layer with unacceptably small activations and increasing its weights, it is possible to eventually obtain a network with reasonable initial activations throughout. If learning is still too slow at this point, it can be useful to look at the range or standard deviation of the gradients as well as the activations. This procedure can in principle be automated and is generally less computationally costly than hyperparameter optimization based on validation set error because it is based on feedback from the behavior of the initial model on a single batch of data, rather than on feedback from a trained model on the validation set. While long used heuristically, this protocol has recently been specified more formally and studied by [Mishkin and Matas \(2015\)](#).

So far we have focused on the initialization of the weights. Fortunately, initialization of other parameters is typically easier.

The approach for setting the biases must be coordinated with the approach for setting the weights. Setting the biases to zero is compatible with most weight initialization schemes. There are a few situations where we may set some biases to nonzero values:

- If a bias is for an output unit, then it is often beneficial to initialize the bias to obtain the right marginal statistics of the output. To do this, we assume that the initial weights are small enough that the output of the unit is determined only by the bias. This justifies setting the bias to the inverse of the activation function applied to the marginal statistics of the output in the training set. For example, if the output is a distribution over classes, and this distribution is a highly skewed distribution with the marginal probability of class i given by element c_i of some vector \mathbf{c} , then we can set the bias vector \mathbf{b} by solving the equation $\text{softmax}(\mathbf{b}) = \mathbf{c}$. This applies not only to classifiers but also to models we will encounter in Part III, such as autoencoders and Boltzmann machines. These models have layers whose output should resemble the input data \mathbf{x} , and it can be very helpful to initialize the biases of such layers to match the marginal distribution over \mathbf{x} .
- Sometimes we may want to choose the bias to avoid causing too much saturation at initialization. For example, we may set the bias of a ReLU hidden unit to 0.1 rather than 0 to avoid saturating the ReLU at initialization. This approach is not compatible with weight initialization schemes that do

not expect strong input from the biases though. For example, it is not recommended for use with random walk initialization (Sussillo, 2014).

- Sometimes a unit controls whether other units are able to participate in a function. In such situations, we have a unit with output u and another unit $h \in [0, 1]$, and they are multiplied together to produce an output uh . We can view h as a gate that determines whether $uh \approx u$ or $uh \approx 0$. In these situations, we want to set the bias for h so that $h \approx 1$ most of the time at initialization. Otherwise u does not have a chance to learn. For example, Jozefowicz *et al.* (2015) advocate setting the bias to 1 for the forget gate of the LSTM model, described in section 10.10.

Another common type of parameter is a variance or precision parameter. For example, we can perform linear regression with a conditional variance estimate using the model

$$p(y | \mathbf{x}) = \mathcal{N}(y | \mathbf{w}^T \mathbf{x} + b, 1/\beta), \quad (8.24)$$

where β is a precision parameter. We can usually initialize variance or precision parameters to 1 safely. Another approach is to assume the initial weights are close enough to zero that the biases may be set while ignoring the effect of the weights, then set the biases to produce the correct marginal mean of the output, and set the variance parameters to the marginal variance of the output in the training set.

Besides these simple constant or random methods of initializing model parameters, it is possible to initialize model parameters using machine learning. A common strategy discussed in part III of this book is to initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs. One can also perform supervised training on a related task. Even performing supervised training on an unrelated task can sometimes yield an initialization that offers faster convergence than a random initialization. Some of these initialization strategies may yield faster convergence and better generalization because they encode information about the distribution in the initial parameters of the model. Others apparently perform well primarily because they set the parameters to have the right scale or set different units to compute different functions from each other.

8.5 Algorithms with Adaptive Learning Rates

Neural network researchers have long realized that the learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance. As we discuss in sections 4.3 and 8.2, the cost is often highly

sensitive to some directions in parameter space and insensitive to others. The momentum algorithm can mitigate these issues somewhat, but it does so at the expense of introducing another hyperparameter. In the face of this, it is natural to ask if there is another way. If we believe that the directions of sensitivity are somewhat axis aligned, it can make sense to use a separate learning rate for each parameter and automatically adapt these learning rates throughout the course of learning.

The **delta-bar-delta** algorithm ([Jacobs, 1988](#)) is an early heuristic approach to adapting individual learning rates for model parameters during training. The approach is based on a simple idea: if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If that partial derivative changes sign, then the learning rate should decrease. Of course, this kind of rule can only be applied to full batch optimization.

More recently, a number of incremental (or mini batch-based) methods have been introduced that adapt the learning rates of model parameters. In this section, we briefly review a few of these algorithms.

8.5.1 AdaGrad

The **AdaGrad** algorithm, shown in algorithm 8.4, individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient ([Duchi et al., 2011](#)). The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties. Empirically, however, for training deep neural network models, the accumulation of squared gradients *from the beginning of training* can result in a premature and excessive decrease in the effective learning rate. AdaGrad performs well for some but not all deep learning models.

8.5.2 RMSProp

The **RMSProp** algorithm ([Hinton, 2012](#)) modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average. AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a nonconvex function to train a neural network,

the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl. AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter θ

Require: Small constant δ , perhaps 10^{-7} , for numerical stability

Initialize gradient accumulation variable $r = \mathbf{0}$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

 Accumulate squared gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$.

 Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$. (Division and square root applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta \theta$.

end while

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ , decay rate ρ

Require: Initial parameter θ

Require: Small constant δ , usually 10^{-6} , used to stabilize division by small numbers

Initialize accumulation variables $r = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

 Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$.

 Compute parameter update: $\Delta \theta = -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{\delta + \mathbf{r}}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta \theta$.

end while

RMSProp is shown in its standard form in algorithm 8.5 and combined with Nesterov momentum in algorithm 8.6. Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ , that controls the length scale of the moving average.

Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

8.5.3 Adam

Adam (Kingma and Ba, 2014) is yet another adaptive learning rate optimization algorithm and is presented in algorithm 8.7. The name “Adam” derives from the phrase “adaptive moments.” In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin (see algorithm 8.7). RMSProp also incorporates an estimate of the

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ , decay rate ρ , momentum coefficient α

Require: Initial parameter θ , initial velocity v

Initialize accumulation variable $r = \mathbf{0}$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$.

 Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) \mathbf{g} \odot \mathbf{g}$.

 Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{r}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + v$.

end while

(uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

8.5.4 Choosing the Right Optimization Algorithm

We have discussed a series of related algorithms that each seek to address the challenge of optimizing deep models by adapting the learning rate for each model parameter. At this point, a natural question is: which algorithm should one choose?

Unfortunately, there is currently no consensus on this point. Schaul *et al.* (2014) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta)

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1]$.
(Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})

Require: Initial parameters θ

Initialize 1st and 2nd moment variables $s = \mathbf{0}$, $r = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

 Update biased first moment estimate: $\hat{s} \leftarrow \rho_1 s + (1 - \rho_1)\mathbf{g}$

 Update biased second moment estimate: $\hat{r} \leftarrow \rho_2 r + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$

 Correct bias in first moment: $\hat{s} \leftarrow \frac{\hat{s}}{1 - \rho_1^t}$

 Correct bias in second moment: $\hat{r} \leftarrow \frac{\hat{r}}{1 - \rho_2^t}$

 Compute update: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$ (operations applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam. The choice of which algorithm to use, at this point, seems to depend largely on the user’s familiarity with the algorithm (for ease of hyperparameter tuning).

8.6 Approximate Second-Order Methods

In this section we discuss the application of second-order methods to training deep networks. See [LeCun *et al.* \(1998a\)](#) for an earlier treatment of this subject. For simplicity of exposition, the only objective function we examine is the empirical risk:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}(\mathbf{x}, y)} [L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (8.25)$$

The methods we discuss here extend readily, however, to more general objective functions, such as those that include parameter regularization terms, as discussed in chapter [7](#).

8.6.1 Newton’s Method

In section [4.3](#), we introduced second-order gradient methods. In contrast to first-order methods, second-order methods make use of second derivatives to improve optimization. The most widely used second-order method is Newton’s method. We now describe Newton’s method in more detail, with emphasis on its application to neural network training.

Newton’s method is an optimization scheme based on using a second-order Taylor series expansion to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0), \quad (8.26)$$

where \mathbf{H} is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$. If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \quad (8.27)$$

Thus for a locally quadratic function (with positive definite \mathbf{H}), by rescaling the gradient by \mathbf{H}^{-1} , Newton's method jumps directly to the minimum. If the objective function is convex but not quadratic (there are higher-order terms), this update can be iterated, yielding the training algorithm associated with Newton's method, given in algorithm 8.8.

For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step iterative procedure. First, update or compute the inverse Hessian (i.e., by updating the quadratic approximation). Second, update the parameters according to equation 8.27.

In section 8.2.3, we discuss how Newton's method is appropriate only when the Hessian is positive definite. In deep learning, the surface of the objective function is typically nonconvex, with many features, such as saddle points, that are problematic for Newton's method. If the eigenvalues of the Hessian are not all positive, for example, near a saddle point, then Newton's method can actually cause updates to move in the wrong direction. This situation can be avoided by regularizing the Hessian. Common regularization strategies include adding a constant, α , along the diagonal of the Hessian. The regularized update becomes

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [H(f(\boldsymbol{\theta}_0)) + \alpha \mathbf{I}]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0). \quad (8.28)$$

This regularization strategy is used in approximations to Newton's method, such as the Levenberg-Marquardt algorithm (Levenberg, 1944; Marquardt, 1963), and works fairly well as long as the negative eigenvalues of the Hessian are still relatively close to zero. When there are more extreme directions of curvature, the value of α would have to be sufficiently large to offset the negative eigenvalues. As α increases in size, however, the Hessian becomes dominated by the $\alpha \mathbf{I}$ diagonal,

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$

Require: Initial parameter $\boldsymbol{\theta}_0$

Require: Training set of m examples

while stopping criterion not met **do**

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$

 Compute Hessian: $\mathbf{H} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}}^2 \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$

 Compute Hessian inverse: \mathbf{H}^{-1}

 Compute update: $\Delta \boldsymbol{\theta} = -\mathbf{H}^{-1} \mathbf{g}$

 Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

and the direction chosen by Newton’s method converges to the standard gradient divided by α . When strong negative curvature is present, α may need to be so large that Newton’s method would make smaller steps than gradient descent with a properly chosen learning rate.

Beyond the challenges created by certain features of the objective function, such as saddle points, the application of Newton’s method for training large neural networks is limited by the significant computational burden it imposes. The number of elements in the Hessian is squared in the number of parameters, so with k parameters (and for even very small neural networks, the number of parameters k can be in the millions), Newton’s method would require the inversion of a $k \times k$ matrix—with computational complexity of $O(k^3)$. Also, since the parameters will change with every update, the inverse Hessian has to be computed *at every training iteration*. As a consequence, only networks with a very small number of parameters can be practically trained via Newton’s method. In the remainder of this section, we discuss alternatives that attempt to gain some of the advantages of Newton’s method while side-stepping the computational hurdles.

8.6.2 Conjugate Gradients

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending **conjugate directions**. The inspiration for this approach follows from a careful study of the weakness of the method of steepest descent (see section 4.3 for details), where line searches are applied iteratively in the direction associated with the gradient. Figure 8.6 illustrates how the method of steepest descent, when applied in a quadratic bowl, progresses in a rather ineffective back-and-forth zig-zag pattern. This happens because each line search direction, when given by the gradient, is guaranteed to be orthogonal to the previous line search direction.

Let the previous search direction be \mathbf{d}_{t-1} . At the minimum, where the line search terminates, the directional derivative is zero in direction \mathbf{d}_{t-1} : $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \mathbf{d}_{t-1} = 0$. Since the gradient at this point defines the current search direction, $\mathbf{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ will have no contribution in the direction \mathbf{d}_{t-1} . Thus \mathbf{d}_t is orthogonal to \mathbf{d}_{t-1} . This relationship between \mathbf{d}_{t-1} and \mathbf{d}_t is illustrated in figure 8.6 for multiple iterations of steepest descent. As demonstrated in the figure, the choice of orthogonal directions of descent do not preserve the minimum along the previous search directions. This gives rise to the zig-zag pattern of progress, where by descending to the minimum in the current gradient direction, we must reminimize the objective in the previous gradient direction. Thus, by following the gradient at the end of each line search we are, in a sense, undoing progress we have already

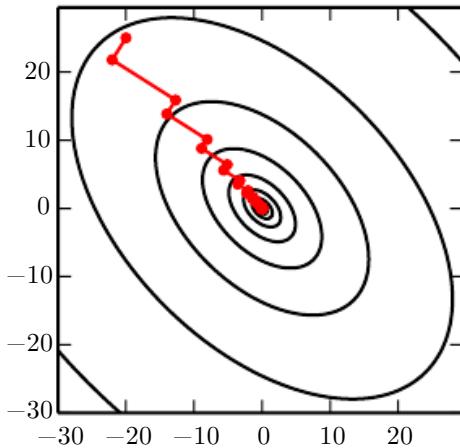


Figure 8.6: The method of steepest descent applied to a quadratic cost surface. The method of steepest descent involves jumping to the point of lowest cost along the line defined by the gradient at the initial point on each step. This resolves some of the problems seen with using a fixed learning rate in figure 4.6, but even with the optimal step size the algorithm still makes back-and-forth progress toward the optimum. By definition, at the minimum of the objective along a given direction, the gradient at the final point is orthogonal to that direction.

made in the direction of the previous line search. The method of conjugate gradients seeks to address this problem.

In the method of conjugate gradients, we seek to find a search direction that is **conjugate** to the previous line search direction; that is, it will not undo progress made in that direction. At training iteration t , the next search direction \mathbf{d}_t takes the form:

$$\mathbf{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \mathbf{d}_{t-1}, \quad (8.29)$$

where β_t is a coefficient whose magnitude controls how much of the direction, \mathbf{d}_{t-1} , we should add back to the current search direction.

Two directions, \mathbf{d}_t and \mathbf{d}_{t-1} , are defined as conjugate if $\mathbf{d}_t^\top \mathbf{H} \mathbf{d}_{t-1} = 0$, where \mathbf{H} is the Hessian matrix.

The straightforward way to impose conjugacy would involve calculation of the eigenvectors of \mathbf{H} to choose β_t , which would not satisfy our goal of developing a method that is more computationally viable than Newton's method for large problems. Can we calculate the conjugate directions without resorting to these calculations? Fortunately, the answer to that is yes.

Two popular methods for computing the β_t are

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\theta} J(\theta_t)^\top \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1})^\top \nabla_{\theta} J(\theta_{t-1})} \quad (8.30)$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\theta} J(\theta_t) - \nabla_{\theta} J(\theta_{t-1}))^\top \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1})^\top \nabla_{\theta} J(\theta_{t-1})} \quad (8.31)$$

For a quadratic surface, the conjugate directions ensure that the gradient along the previous direction does not increase in magnitude. We therefore stay at the minimum along the previous directions. As a consequence, in a k -dimensional parameter space, the conjugate gradient method requires at most k line searches to achieve the minimum. The conjugate gradient algorithm is given in algorithm 8.9.

Algorithm 8.9 The conjugate gradient method

Require: Initial parameters θ_0

Require: Training set of m examples

Initialize $\rho_0 = \mathbf{0}$

Initialize $g_0 = 0$

Initialize $t = 1$

while stopping criterion not met **do**

 Initialize the gradient $\mathbf{g}_t = \mathbf{0}$

 Compute gradient: $\mathbf{g}_t \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Compute $\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^\top \mathbf{g}_t}{\mathbf{g}_{t-1}^\top \mathbf{g}_{t-1}}$ (Polak-Ribière)

 (Nonlinear conjugate gradient: optionally reset β_t to zero, for example if t is a multiple of some constant k , such as $k = 5$)

 Compute search direction: $\rho_t = -\mathbf{g}_t + \beta_t \rho_{t-1}$

 Perform line search to find: $\epsilon^* = \operatorname{argmin}_{\epsilon} \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \theta_t + \epsilon \rho_t), \mathbf{y}^{(i)})$

 (On a truly quadratic cost function, analytically solve for ϵ^* rather than explicitly searching for it)

 Apply update: $\theta_{t+1} = \theta_t + \epsilon^* \rho_t$

$t \leftarrow t + 1$

end while

Nonlinear Conjugate Gradients: So far we have discussed the method of conjugate gradients as it is applied to quadratic objective functions. Of course, our primary interest in this chapter is to explore optimization methods for training neural networks and other related deep learning models where the corresponding

objective function is far from quadratic. Perhaps surprisingly, the method of conjugate gradients is still applicable in this setting, though with some modification. Without any assurance that the objective is quadratic, the conjugate directions are no longer assured to remain at the minimum of the objective for previous directions. As a result, the **nonlinear conjugate gradients** algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient.

Practitioners report reasonable results in applications of the nonlinear conjugate gradients algorithm to training neural networks, though it is often beneficial to initialize the optimization with a few iterations of stochastic gradient descent before commencing nonlinear conjugate gradients. Also, while the (nonlinear) conjugate gradients algorithm has traditionally been cast as a batch method, minibatch versions have been used successfully for training neural networks (Le *et al.*, 2011). Adaptations of conjugate gradients specifically for neural networks have been proposed earlier, such as the scaled conjugate gradients algorithm (Moller, 1993).

8.6.3 BFGS

The **Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm** attempts to bring some of the advantages of Newton’s method without the computational burden. In that respect, BFGS is similar to the conjugate gradient method. However, BFGS takes a more direct approach to the approximation of Newton’s update. Recall that Newton’s update is given by

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \quad (8.32)$$

where \mathbf{H} is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$. The primary computational difficulty in applying Newton’s update is the calculation of the inverse Hessian \mathbf{H}^{-1} . The approach adopted by quasi-Newton methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix \mathbf{M}_t that is iteratively refined by low-rank updates to become a better approximation of \mathbf{H}^{-1} .

The specification and derivation of the BFGS approximation is given in many textbooks on optimization, including in Luenberger (1984).

Once the inverse Hessian approximation \mathbf{M}_t is updated, the direction of descent $\boldsymbol{\rho}_t$ is determined by $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$. A line search is performed in this direction to determine the size of the step, ϵ^* , taken in this direction. The final update to the parameters is given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t. \quad (8.33)$$

Like the method of conjugate gradients, the BFGS algorithm iterates a series of line searches with the direction incorporating second-order information. Unlike conjugate gradients, however, the success of the approach is not heavily dependent on the line search finding a point very close to the true minimum along the line. Thus, relative to conjugate gradients, BFGS has the advantage that it can spend less time refining each line search. On the other hand, the BFGS algorithm must store the inverse Hessian matrix, \mathbf{M} , that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters.

Limited Memory BFGS (or L-BFGS) The memory costs of the BFGS algorithm can be significantly decreased by avoiding storing the complete inverse Hessian approximation \mathbf{M} . The L-BFGS algorithm computes the approximation \mathbf{M} using the same method as the BFGS algorithm but beginning with the assumption that $\mathbf{M}^{(t-1)}$ is the identity matrix, rather than storing the approximation from one step to the next. If used with exact line searches, the directions defined by L-BFGS are mutually conjugate. However, unlike the method of conjugate gradients, this procedure remains well behaved when the minimum of the line search is reached only approximately. The L-BFGS strategy with no storage described here can be generalized to include more information about the Hessian by storing some of the vectors used to update \mathbf{M} at each time step, which costs only $O(n)$ per step.

8.7 Optimization Strategies and Meta-Algorithms

Many optimization techniques are not exactly algorithms but rather general templates that can be specialized to yield algorithms, or subroutines that can be incorporated into many different algorithms.

8.7.1 Batch Normalization

Batch normalization (Ioffe and Szegedy, 2015) is one of the most exciting recent innovations in optimizing deep neural networks, and it is actually not an optimization algorithm at all. Instead, it is a method of adaptive reparametrization, motivated by the difficulty of training very deep models.

Very deep models involve the composition of several functions, or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all the layers simultaneously. When we make the update, unexpected results can happen because many functions composed

together are changed simultaneously, using updates that were computed under the assumption that the other functions remain constant. As a simple example, suppose we have a deep neural network that has only one unit per layer and does not use an activation function at each hidden layer: $\hat{y} = xw_1w_2w_3 \dots w_l$. Here, w_i provides the weight used by layer i . The output of layer i is $h_i = h_{i-1}w_i$. The output \hat{y} is a linear function of the input x but a nonlinear function of the weights w_i . Suppose our cost function has put a gradient of 1 on \hat{y} , so we wish to decrease \hat{y} slightly. The back-propagation algorithm can then compute a gradient $\mathbf{g} = \nabla_{\mathbf{w}}\hat{y}$. Consider what happens when we make an update $\mathbf{w} \leftarrow \mathbf{w} - \epsilon\mathbf{g}$. The first-order Taylor series approximation of \hat{y} predicts that the value of \hat{y} will decrease by $\epsilon\mathbf{g}^\top\mathbf{g}$. If we wanted to decrease \hat{y} by 0.1, this first-order information available in the gradient suggests we could set the learning rate ϵ to $\frac{0.1}{\mathbf{g}^\top\mathbf{g}}$. Yet, the actual update will include second-order and third-order effects, on up to effects of order l . The new value of \hat{y} is given by

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l). \quad (8.34)$$

An example of one second-order term arising from this update is $\epsilon^2 g_1 g_2 \prod_{i=3}^l w_i$. This term might be negligible if $\prod_{i=3}^l w_i$ is small, or might be exponentially large if the weights on layers 3 through l are greater than 1. This makes it very hard to choose an appropriate learning rate, because the effects of an update to the parameters for one layer depend so strongly on all the other layers. Second-order optimization algorithms address this issue by computing an update that takes these second-order interactions into account, but we can see that in very deep networks, even higher-order interactions can be significant. Even second-order optimization algorithms are expensive and usually require numerous approximations that prevent them from truly accounting for all significant second-order interactions. Building an n -th order optimization algorithm for $n > 2$ thus seems hopeless. What can we do instead?

Batch normalization provides an elegant way of reparametrizing almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers. Batch normalization can be applied to any input or hidden layer in a network. Let \mathbf{H} be a minibatch of activations of the layer to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix. To normalize \mathbf{H} , we replace it with

$$\mathbf{H}' = \frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \quad (8.35)$$

where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit. The arithmetic here is based on broadcasting

the vector $\boldsymbol{\mu}$ and the vector $\boldsymbol{\sigma}$ to be applied to every row of the matrix \mathbf{H} . Within each row, the arithmetic is element-wise, so $H_{i,j}$ is normalized by subtracting μ_j and dividing by σ_j . The rest of the network then operates on \mathbf{H}' in exactly the same way that the original network operated on \mathbf{H} .

At training time,

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \mathbf{H}_{i,:} \quad (8.36)$$

and

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\mathbf{H} - \boldsymbol{\mu})_i^2}, \quad (8.37)$$

where δ is a small positive value such as 10^{-8} , imposed to avoid encountering the undefined gradient of \sqrt{z} at $z = 0$. Crucially, we *back-propagate through these operations* for computing the mean and the standard deviation, and for applying them to normalize \mathbf{H} . This means that the gradient will never propose an operation that acts simply to increase the standard deviation or mean of h_i ; the normalization operations remove the effect of such an action and zero out its component in the gradient. This was a major innovation of the batch normalization approach. Previous approaches had involved adding penalties to the cost function to encourage units to have normalized activation statistics or involved intervening to renormalize unit statistics after each gradient descent step. The former approach usually resulted in imperfect normalization and the latter usually resulted in significant wasted time, as the learning algorithm repeatedly proposed changing the mean and variance, and the normalization step repeatedly undid this change. Batch normalization reparametrizes the model to make some units always be standardized by definition, deftly sidestepping both problems.

At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ may be replaced by running averages that were collected during training time. This allows the model to be evaluated on a single example, without needing to use definitions of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ that depend on an entire minibatch.

Revisiting the $\hat{y} = xw_1w_2 \dots w_l$ example, we see that we can mostly resolve the difficulties in learning this model by normalizing h_{l-1} . Suppose that x is drawn from a unit Gaussian. Then h_{l-1} will also come from a Gaussian, because the transformation from x to h_l is linear. However, h_{l-1} will no longer have zero mean and unit variance. After applying batch normalization, we obtain the normalized \hat{h}_{l-1} that restores the zero mean and unit variance properties. For almost any update to the lower layers, \hat{h}_{l-1} will remain a unit Gaussian. The output \hat{y} may then be learned as a simple linear function $\hat{y} = w_l \hat{h}_{l-1}$. Learning in this model is now very simple because the parameters at the lower layers do not have an effect in most cases; their output is always renormalized to a unit Gaussian. In some

corner cases, the lower layers can have an effect. Changing one of the lower layer weights to 0 can make the output become degenerate, and changing the sign of one of the lower weights can flip the relationship between \hat{h}_{l-1} and y . These situations are very rare. Without normalization, nearly every update would have an extreme effect on the statistics of h_{l-1} . Batch normalization has thus made this model significantly easier to learn. In this example, the ease of learning of course came at the cost of making the lower layers useless. In our linear example, the lower layers no longer have any harmful effect, but they also no longer have any beneficial effect. This is because we have normalized out the first- and second-order statistics, which is all that a linear network can influence. In a deep neural network with nonlinear activation functions, the lower layers can perform nonlinear transformations of the data, so they remain useful. Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, but it allows the relationships between units and the nonlinear statistics of a single unit to change.

Because the final layer of the network is able to learn a linear transformation, we may actually wish to remove all linear relationships between units within a layer. Indeed, this is the approach taken by Desjardins *et al.* (2015), who provided the inspiration for batch normalization. Unfortunately, eliminating all linear interactions is much more expensive than standardizing the mean and standard deviation of each individual unit, and so far batch normalization remains the most practical approach.

Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network containing that unit. To maintain the expressive power of the network, it is common to replace the batch of hidden unit activations \mathbf{H} with $\gamma\mathbf{H}' + \beta$ rather than simply the normalized \mathbf{H}' . The variables γ and β are learned parameters that allow the new variable to have any mean and standard deviation. At first glance, this may seem useless—why did we set the mean to $\mathbf{0}$, and then introduce a parameter that allows it to be set back to any arbitrary value β ? The answer is that the new parametrization can represent the same family of functions of the input as the old parametrization, but the new parametrization has different learning dynamics. In the old parametrization, the mean of \mathbf{H} was determined by a complicated interaction between the parameters in the layers below \mathbf{H} . In the new parametrization, the mean of $\gamma\mathbf{H}' + \beta$ is determined solely by β . The new parametrization is much easier to learn with gradient descent.

Most neural network layers take the form of $\phi(\mathbf{XW} + \mathbf{b})$, where ϕ is some fixed nonlinear activation function such as the rectified linear transformation. It is natural to wonder whether we should apply batch normalization to the input \mathbf{X} , or to the transformed value $\mathbf{XW} + \mathbf{b}$. Ioffe and Szegedy (2015) recommend

the latter. More specifically, $\mathbf{X}\mathbf{W} + \mathbf{b}$ should be replaced by a normalized version of $\mathbf{X}\mathbf{W}$. The bias term should be omitted because it becomes redundant with the β parameter applied by the batch normalization reparametrization. The input to a layer is usually the output of a nonlinear activation function such as the rectified linear function in a previous layer. The statistics of the input are thus more non-Gaussian and less amenable to standardization by linear operations.

In convolutional networks, described in chapter 9, it is important to apply the same normalizing μ and σ at every spatial location within a feature map, so that the statistics of the feature map remain the same regardless of spatial location.

8.7.2 Coordinate Descent

In some cases, it may be possible to solve an optimization problem quickly by breaking it into separate pieces. If we minimize $f(\mathbf{x})$ with respect to a single variable x_i , then minimize it with respect to another variable x_j , and so on, repeatedly cycling through all variables, we are guaranteed to arrive at a (local) minimum. This practice is known as **coordinate descent**, because we optimize one coordinate at a time. More generally, **block coordinate descent** refers to minimizing with respect to a subset of the variables simultaneously. The term “coordinate descent” is often used to refer to block coordinate descent as well as the strictly individual coordinate descent.

Coordinate descent makes the most sense when the different variables in the optimization problem can be clearly separated into groups that play relatively isolated roles, or when optimization with respect to one group of variables is significantly more efficient than optimization with respect to all of the variables. For example, consider the cost function

$$J(\mathbf{H}, \mathbf{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (\mathbf{X} - \mathbf{W}^\top \mathbf{H})_{i,j}^2. \quad (8.38)$$

This function describes a learning problem called sparse coding, where the goal is to find a weight matrix \mathbf{W} that can linearly decode a matrix of activation values \mathbf{H} to reconstruct the training set \mathbf{X} . Most applications of sparse coding also involve weight decay or a constraint on the norms of the columns of \mathbf{W} , to prevent the pathological solution with extremely small \mathbf{H} and large \mathbf{W} .

The function J is not convex. However, we can divide the inputs to the training algorithm into two sets: the dictionary parameters \mathbf{W} and the code representations \mathbf{H} . Minimizing the objective function with respect to either one of these sets of variables is a convex problem. Block coordinate descent thus gives

us an optimization strategy that allows us to use efficient convex optimization algorithms, by alternating between optimizing \mathbf{W} with \mathbf{H} fixed, then optimizing \mathbf{H} with \mathbf{W} fixed.

Coordinate descent is not a very good strategy when the value of one variable strongly influences the optimal value of another variable, as in the function $f(\mathbf{x}) = (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$ where α is a positive constant. The first term encourages the two variables to have similar value, while the second term encourages them to be near zero. The solution is to set both to zero. Newton's method can solve the problem in a single step because it is a positive definite quadratic problem. For small α , however, coordinate descent will make very slow progress because the first term does not allow a single variable to be changed to a value that differs significantly from the current value of the other variable.

8.7.3 Polyak Averaging

Polyak averaging (Polyak and Juditsky, 1992) consists of averaging several points in the trajectory through parameter space visited by an optimization algorithm. If t iterations of gradient descent visit points $\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(t)}$, then the output of the Polyak averaging algorithm is $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t} \sum_i \boldsymbol{\theta}^{(i)}$. On some problem classes, such as gradient descent applied to convex problems, this approach has strong convergence guarantees. When applied to neural networks, its justification is more heuristic, but it performs well in practice. The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all the locations on either side should be close to the bottom of the valley though.

In nonconvex problems, the path taken by the optimization trajectory can be very complicated and visit many different regions. Including points in parameter space from the distant past that may be separated from the current point by large barriers in the cost function does not seem like a useful behavior. As a result, when applying Polyak averaging to nonconvex problems, it is typical to use an exponentially decaying running average:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha \hat{\boldsymbol{\theta}}^{(t-1)} + (1 - \alpha) \boldsymbol{\theta}^{(t)}. \quad (8.39)$$

The running average approach is used in numerous applications. See Szegedy *et al.* (2015) for a recent example.

8.7.4 Supervised Pretraining

Sometimes, directly training a model to solve a specific task can be too ambitious if the model is complex and hard to optimize or if the task is very difficult. It is sometimes more effective to train a simpler model to solve the task, then make the model more complex. It can also be more effective to train the model to solve a simpler task, then move on to confront the final task. These strategies that involve training simple models on simple tasks before confronting the challenge of training the desired model to perform the desired task are collectively known as **pretraining**.

Greedy algorithms break a problem into many components, then solve for the optimal version of each component in isolation. Unfortunately, combining the individually optimal components is not guaranteed to yield an optimal complete solution. Nonetheless, greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution, and the quality of a greedy solution is often acceptable if not optimal. Greedy algorithms may also be followed by a **fine-tuning** stage in which a joint optimization algorithm searches for an optimal solution to the full problem. Initializing the joint optimization algorithm with a greedy solution can greatly speed it up and improve the quality of the solution it finds.

Pretraining, and especially greedy pretraining, algorithms are ubiquitous in deep learning. In this section, we describe specifically those pretraining algorithms that break supervised learning problems into other simpler supervised learning problems. This approach is known as **greedy supervised pretraining**.

In the original ([Bengio et al., 2007](#)) version of greedy supervised pretraining, each stage consists of a supervised learning training task involving only a subset of the layers in the final neural network. An example of greedy supervised pretraining is illustrated in figure 8.7, in which each added hidden layer is pretrained as part of a shallow supervised MLP, taking as input the output of the previously trained hidden layer. Instead of pretraining one layer at a time, ([Simonyan and Zisserman \(2015\)](#) pretrain a deep convolutional network (eleven weight layers) and then use the first four and last three layers from this network to initialize even deeper networks (with up to nineteen layers of weights). The middle layers of the new, very deep network are initialized randomly. The new network is then jointly trained. Another option, explored by ([Yu et al. \(2010\)](#)), is to use the *outputs* of the previously trained MLPs, as well as the raw input, as inputs for each added stage.

Why would greedy supervised pretraining help? The hypothesis initially discussed by ([Bengio et al. \(2007\)](#)) is that it helps to provide better guidance to the

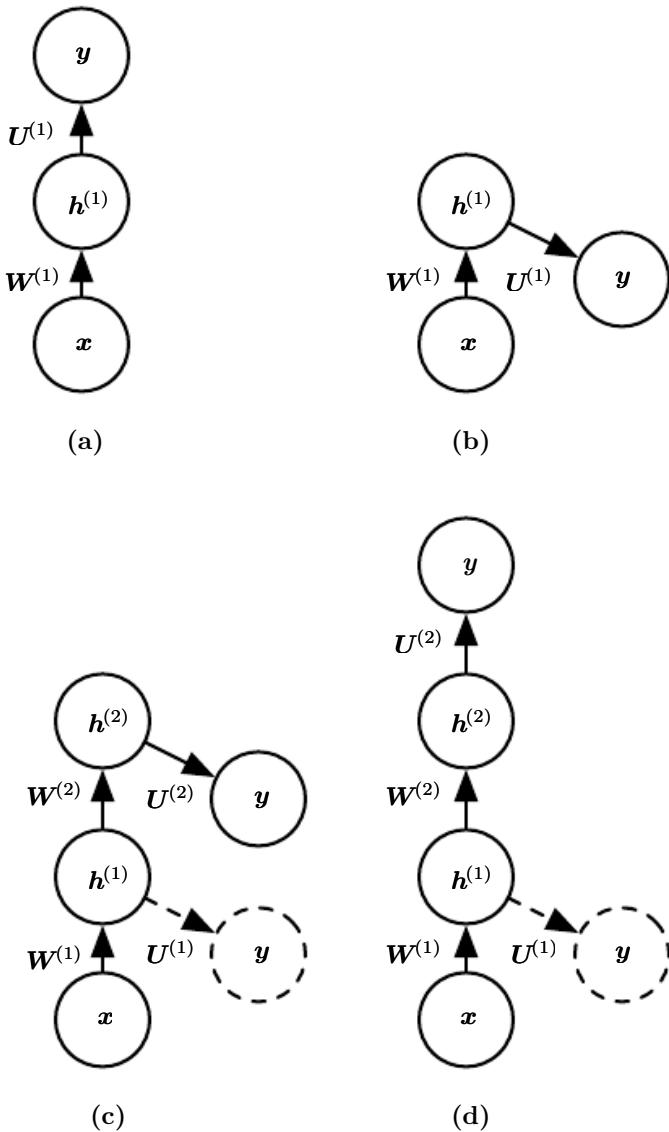


Figure 8.7: Illustration of one form of greedy supervised pretraining (Bengio *et al.*, 2007). (a) We start by training a sufficiently shallow architecture. (b) Another drawing of the same architecture. (c) We keep only the input-to-hidden layer of the original network and discard the hidden-to-output layer. We send the output of the first hidden layer as input to another supervised single hidden layer MLP that is trained with the same objective as the first network was, thus adding a second hidden layer. This can be repeated for as many layers as desired. (d) Another drawing of the result, viewed as a feedforward network. To further improve the optimization, we can jointly fine-tune all the layers, either only at the end or at each stage of this process.

intermediate levels of a deep hierarchy. In general, pretraining may help both in terms of optimization and generalization.

An approach related to supervised pretraining extends the idea to the context of transfer learning: [Yosinski *et al.* \(2014\)](#) pretrain a deep convolutional net with eight layers of weights on a set of tasks (a subset of the 1,000 ImageNet object categories) and then initialize a same-size network with the first k layers of the first net. All the layers of the second network (with the upper layers initialized randomly) are then jointly trained to perform a different set of tasks (another subset of the 1,000 ImageNet object categories), with fewer training examples than for the first set of tasks. Other approaches to transfer learning with neural networks are discussed in section [15.2](#).

Another related line of work is the **FitNets** ([Romero *et al.*, 2015](#)) approach. This approach begins by training a network that has low enough depth and great enough width (number of units per layer) to be easy to train. This network then becomes a **teacher** for a second network, designated the **student**. The student network is much deeper and thinner (eleven to nineteen layers) and would be difficult to train with SGD under normal circumstances. The training of the student network is made easier by training the student network not only to predict the output for the original task, but also to predict the value of the middle layer of the teacher network. This extra task provides a set of hints about how the hidden layers should be used and can simplify the optimization problem. Additional parameters are introduced to regress the middle layer of the five layer teacher network from the middle layer of the deeper student network. Instead of predicting the final classification target, however, the objective is to predict the middle hidden layer of the teacher network. The lower layers of the student networks thus have two objectives: to help the outputs of the student network accomplish their task, as well as to predict the intermediate layer of the teacher network. Although a thin and deep network appears to be more difficult to train than a wide and shallow network, the thin and deep network may generalize better and certainly has lower computational cost if it is thin enough to have far fewer parameters. Without the hints on the hidden layer, the student network performs very poorly in the experiments, on both the training and the test set. Hints on middle layers may thus be one of the tools to help train neural networks that otherwise seem difficult to train, but other optimization techniques or changes in the architecture may also solve the problem.

8.7.5 Designing Models to Aid Optimization

To improve optimization, the best strategy is not always to improve the optimization algorithm. Instead, many improvements in the optimization of deep models have come from designing the models to be easier to optimize.

In principle, we could use activation functions that increase and decrease in jagged nonmonotonic patterns, but this would make optimization extremely difficult. In practice, *it is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm*. Most of the advances in neural network learning over the past thirty years have been obtained by changing the model family rather than changing the optimization procedure. Stochastic gradient descent with momentum, which was used to train neural networks in the 1980s, remains in use in modern state-of-the-art neural network applications.

Specifically, modern neural networks reflect a *design choice* to use linear transformations between layers and activation functions that are differentiable almost everywhere, with significant slope in large portions of their domain. In particular, model innovations like the LSTM, rectified linear units and maxout units have all moved toward using more linear functions than previous models like deep networks based on sigmoidal units. These models have nice properties that make optimization easier. The gradient flows through many layers provided that the Jacobian of the linear transformation has reasonable singular values. Moreover, linear functions consistently increase in a single direction, so even if the model’s output is very far from correct, it is clear simply from computing the gradient which direction its output should move to reduce the loss function. In other words, modern neural nets have been designed so that their *local* gradient information corresponds reasonably well to moving toward a distant solution.

Other model design strategies can help to make optimization easier. For example, linear paths or skip connections between layers reduce the length of the shortest path from the lower layer’s parameters to the output, and thus mitigate the vanishing gradient problem (Srivastava *et al.*, 2015). A related idea to skip connections is adding extra copies of the output that are attached to the intermediate hidden layers of the network, as in GoogLeNet (Szegedy *et al.*, 2014a) and deeply supervised nets (Lee *et al.*, 2014). These “auxiliary heads” are trained to perform the same task as the primary output at the top of the network to ensure that the lower layers receive a large gradient. When training is complete, the auxiliary heads may be discarded. This is an alternative to the pretraining strategies, which were introduced in the previous section. In this way, one can train jointly all the layers in a single phase but change the architecture, so that intermediate layers (especially the lower ones) can get some hints about what they

should do, via a shorter path. These hints provide an error signal to lower layers.

8.7.6 Continuation Methods and Curriculum Learning

As argued in section 8.2.7, many of the challenges in optimization arise from the global structure of the cost function and cannot be resolved merely by making better estimates of local update directions. The predominant strategy for overcoming this problem is to attempt to initialize the parameters in a region connected to the solution by a short path through parameter space that local descent can discover.

Continuation methods are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space. The idea behind continuation methods is to construct a series of objective functions over the same parameters. To minimize a cost function $J(\boldsymbol{\theta})$, we construct new cost functions $\{J^{(0)}, \dots, J^{(n)}\}$. These cost functions are designed to be increasingly difficult, with $J^{(0)}$ being fairly easy to minimize, and $J^{(n)}$, the most difficult, being $J(\boldsymbol{\theta})$, the true cost function motivating the entire process. When we say that $J^{(i)}$ is easier than $J^{(i+1)}$, we mean that it is well behaved over more of $\boldsymbol{\theta}$ space. A random initialization is more likely to land in the region where local descent can minimize the cost function successfully because this region is larger. The series of cost functions are designed so that a solution to one is a good initial point of the next. We thus begin by solving an easy problem, then refine the solution to solve incrementally harder problems until we arrive at a solution to the true underlying problem.

Traditional continuation methods (predating the use of continuation methods for neural network training) are usually based on smoothing the objective function. See Wu (1997) for an example of such a method and a review of some related methods. Continuation methods are also closely related to simulated annealing, which adds noise to the parameters (Kirkpatrick *et al.*, 1983). Continuation methods have been extremely successful in recent years. See Mabahi and Fisher (2015) for an overview of recent literature, especially for AI applications.

Continuation methods traditionally were mostly designed with the goal of overcoming the challenge of local minima. Specifically, they were designed to reach a global minimum despite the presence of many local minima. To do so, these continuation methods would construct easier cost functions by “blurring” the original cost function. This blurring operation can be done by approximating

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}') \quad (8.40)$$

via sampling. The intuition for this approach is that some nonconvex functions become approximately convex when blurred. In many cases, this blurring preserves

enough information about the location of a global minimum that we can find the global minimum by solving progressively less-blurred versions of the problem. This approach can break down in three different ways. First, it might successfully define a series of cost functions where the first is convex and the optimum tracks from one function to the next, arriving at the global minimum, but it might require so many incremental cost functions that the cost of the entire procedure remains high. NP-hard optimization problems remain NP-hard, even when continuation methods are applicable. The other two ways continuation methods fail both correspond to the method not being applicable. First, the function might not become convex, no matter how much it is blurred. Consider, for example, the function $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$. Second, the function may become convex as a result of blurring, but the minimum of this blurred function may track to a local rather than a global minimum of the original cost function.

Though continuation methods were mostly originally designed to deal with the problem of local minima, local minima are no longer believed to be the primary problem for neural network optimization. Fortunately, continuation methods can still help. The easier objective functions introduced by the continuation method can eliminate flat regions, decrease variance in gradient estimates, improve conditioning of the Hessian matrix, or do anything else that will either make local updates easier to compute or improve the correspondence between local update directions and progress toward a global solution.

Bengio *et al.* (2009) observed that an approach called **curriculum learning**, or **shaping**, can be interpreted as a continuation method. Curriculum learning is based on the idea of planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts. This basic strategy was previously known to accelerate progress in animal training (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009) and in machine learning (Solomonoff, 1989; Elman, 1993; Sanger, 1994). Bengio *et al.* (2009) justified this strategy as a continuation method, where earlier $J^{(i)}$ are made easier by increasing the influence of simpler examples (either by assigning their contributions to the cost function larger coefficients, or by sampling them more frequently), and experimentally demonstrated that better results could be obtained by following a curriculum on a large-scale neural language modeling task. Curriculum learning has been successful on a wide range of natural language (Spitkovsky *et al.*, 2010; Collobert *et al.*, 2011a; Mikolov *et al.*, 2011b; Tu and Honavar, 2011) and computer vision (Kumar *et al.*, 2010; Lee and Grauman, 2011; Supancic and Ramanan, 2013) tasks. Curriculum learning was also verified as being consistent with the way in which humans *teach* (Khan *et al.*, 2011): teachers start by showing easier and more prototypical examples and then help the learner refine the decision surface

with the less obvious cases. Curriculum-based strategies are *more effective* for teaching humans than strategies based on uniform sampling of examples and can also increase the effectiveness of other teaching strategies (Basu and Christensen, 2013).

Another important contribution to research on curriculum learning arose in the context of training recurrent neural networks to capture long-term dependencies: Zaremba and Sutskever (2014) found that much better results were obtained with a *stochastic curriculum*, in which a random mix of easy and difficult examples is always presented to the learner, but where the average proportion of the more difficult examples (here, those with longer-term dependencies) is gradually increased. With a deterministic curriculum, no improvement over the baseline (ordinary training from the full training set) was observed.

We have now described the basic family of neural network models and how to regularize and optimize them. In the chapters ahead, we turn to specializations of the neural network family that allow neural networks to scale to very large sizes and process input data that has special structure. The optimization methods discussed in this chapter are often directly applicable to these specialized architectures with little or no modification.

Chapter 9

Convolutional Networks

Convolutional networks (LeCun, 1989), also known as **convolutional neural networks**, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name “convolutional neural network” indicates that the network employs a mathematical operation called **convolution**. Convolution is a specialized kind of linear operation. *Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*

In this chapter, we first describe what convolution is. Next, we explain the motivation behind using convolution in a neural network. We then describe an operation called **pooling**, which almost all convolutional networks employ. Usually, the operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields, such as engineering or pure mathematics. We describe several variants on the convolution function that are widely used in practice for neural networks. We also show how convolution may be applied to many kinds of data, with different numbers of dimensions. We then discuss means of making convolution more efficient. Convolutional networks stand out as an example of neuroscientific principles influencing deep learning. We discuss these neuroscientific principles, then conclude with comments about the role convolutional networks have played in the history of deep learning. One topic this chapter does not address is how to choose the architecture of your convolutional network. The goal of this chapter is to describe the kinds of tools that convolutional networks provide, while chapter 11 describes general guidelines

for choosing which tools to use in which circumstances. Research into convolutional network architectures proceeds so rapidly that a new best architecture for a given benchmark is announced every few weeks to months, rendering it impractical to describe in print the best architecture. Nonetheless, the best architectures have consistently been composed of the building blocks described here.

9.1 The Convolution Operation

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time t . Both x and t are real valued, that is, we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da. \quad (9.1)$$

This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t). \quad (9.2)$$

In our example, w needs to be a valid probability density function, or the output will not be a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example, though. In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input**, and the second

argument (in this example, the function w) as the **kernel**. The output is sometimes referred to as the **feature map**.

In our example, the idea of a laser sensor that can provide measurements at every instant is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t , we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \quad (9.3)$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but in the finite set of points for which we store the values. This means that in practice, we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K :

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n). \quad (9.4)$$

Convolution is commutative, meaning we can equivalently write

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n). \quad (9.5)$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n .

The commutative property of convolution arises because we have **flipped** the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property

is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the **cross-correlation**, which is the same as convolution but without flipping the kernel:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n). \quad (9.6)$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we follow this convention of calling both operations convolution and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel that is flipped relative to the kernel learned by an algorithm without the flipping. It is also rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not.

See figure 9.1 for an example of convolution (without kernel flipping) applied to a 2-D tensor.

Discrete convolution can be viewed as multiplication by a matrix, but the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This is known as a **Toeplitz matrix**. In two dimensions, a **doubly block circulant matrix** corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution, without requiring any further changes to the neural network. Typical convolutional neural networks do make use of further specializations in order to deal with large inputs efficiently, but these are not strictly necessary from a theoretical perspective.

9.2 Motivation

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant**

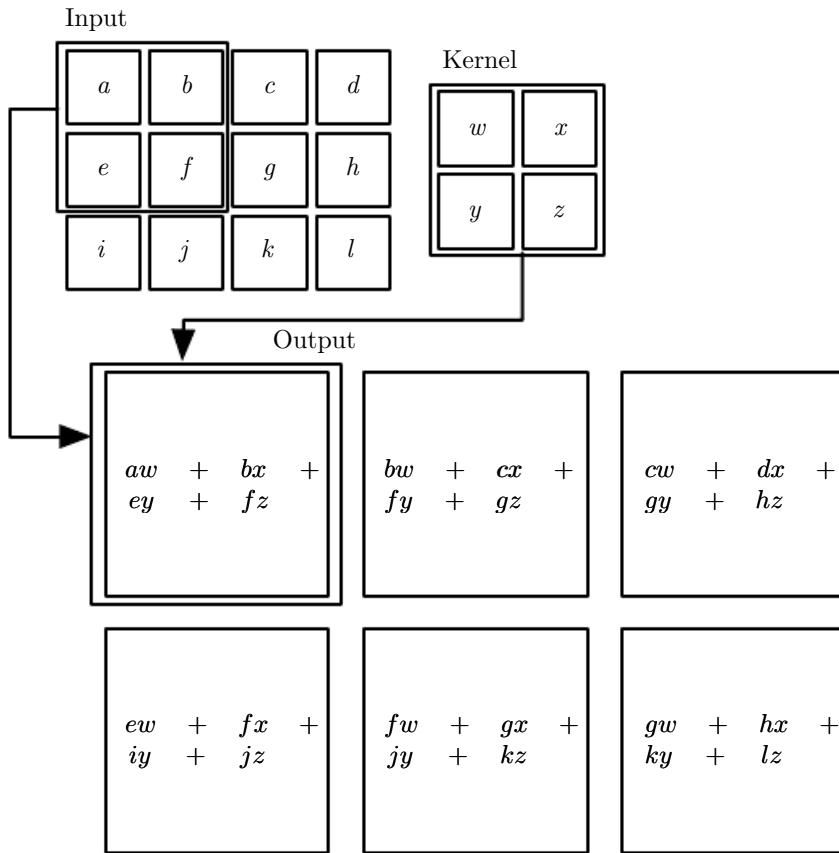


Figure 9.1: An example of 2-D convolution without kernel flipping. We restrict the output to only positions where the kernel lies entirely within the image, called “valid” convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

representations. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have **sparse interactions** (also referred to as **sparse connectivity** or **sparse weights**). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output

requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k , then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m . For graphical demonstrations of sparse connectivity, see figure 9.2 and figure 9.3. In a deep convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input, as shown in figure 9.4. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix

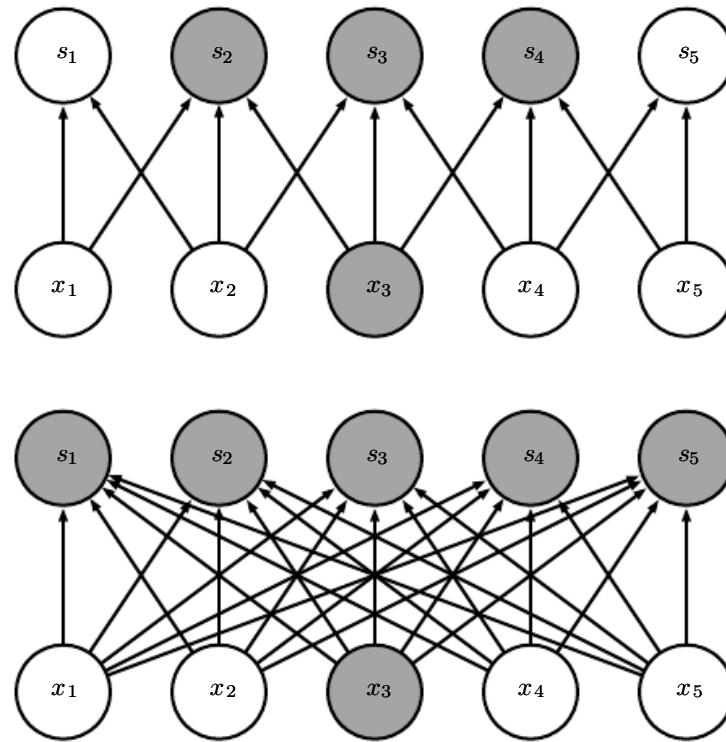


Figure 9.2: Sparse connectivity, viewed from below. We highlight one input unit, x_3 , and highlight the output units in \mathbf{s} that are affected by this unit. (Top) When \mathbf{s} is formed by convolution with a kernel of width 3, only three outputs are affected by \mathbf{x} . (Bottom) When \mathbf{s} is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by x_3 .

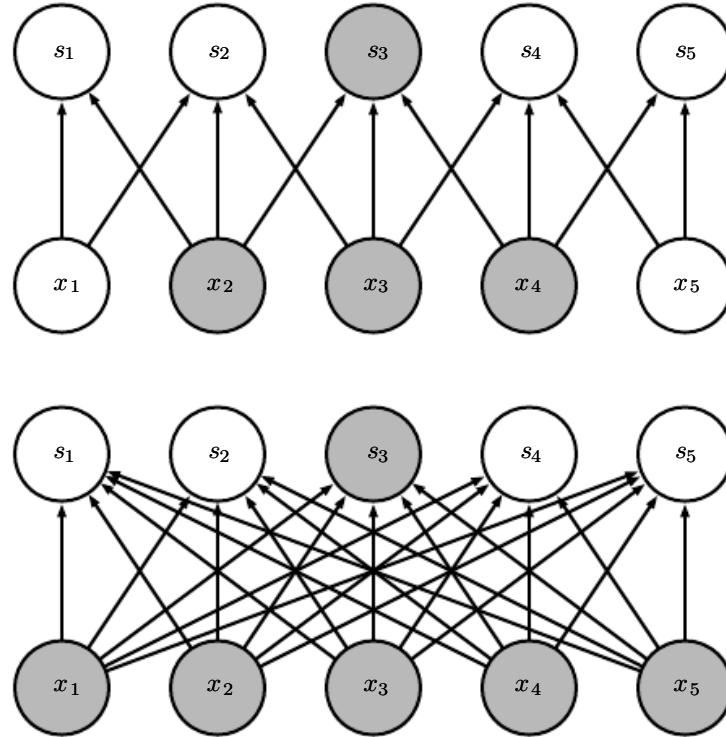


Figure 9.3: Sparse connectivity, viewed from above. We highlight one output unit, s_3 , and highlight the input units in \mathbf{x} that affect this unit. These units are known as the **receptive field** of s_3 . (Top) When \mathbf{s} is formed by convolution with a kernel of width 3, only three inputs affect s_3 . (Bottom) When \mathbf{s} is formed by matrix multiplication, connectivity is no longer sparse, so all the inputs affect s_3 .

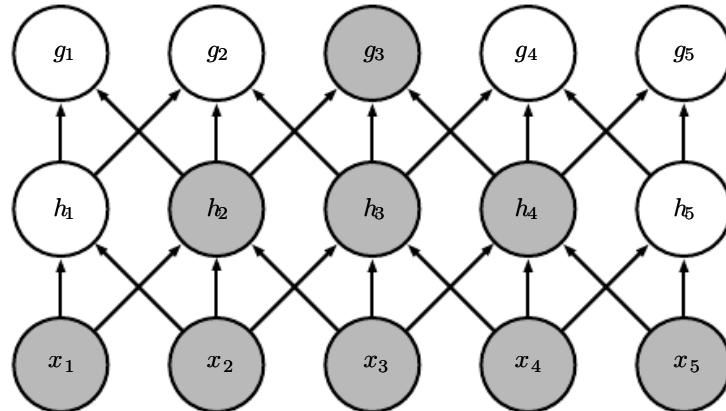


Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (figure 9.12) or pooling (section 9.3). This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has **tied weights**, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still $O(k \times n)$ —but it does further reduce the storage requirements of the model to k parameters. Recall that k is usually several orders of magnitude smaller than m . Since m and n are usually roughly the same size, k is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see figure 9.5.

As an example of both of these first two principles in action, figure 9.6 shows how sparse connectivity and parameter sharing can dramatically improve the

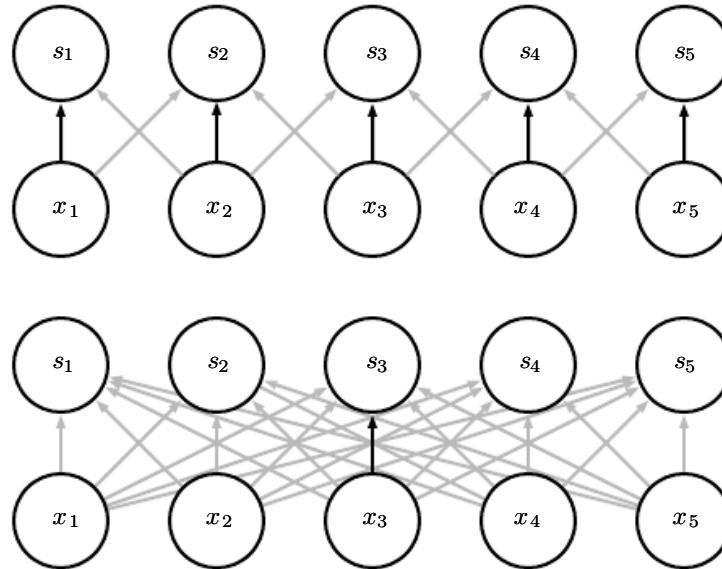


Figure 9.5: Parameter sharing. Black arrows indicate the connections that use a particular parameter in two different models. (Top) The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Because of parameter sharing, this single parameter is used at all input locations. (Bottom) The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing, so the parameter is used only once.

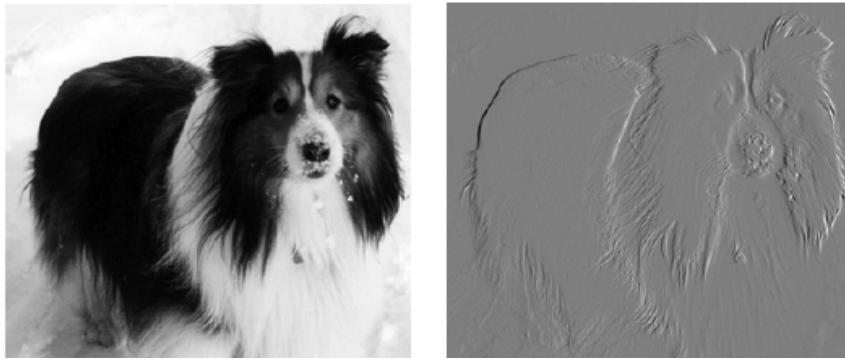


Figure 9.6: Efficiency of edge detection. The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide, while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing two elements, and requires $319 \times 280 \times 3 = 267,960$ floating-point operations (two multiplications and one addition per output pixel) to compute using convolution. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over eight billion, entries in the matrix, making convolution four billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over sixteen billion floating point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating-point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small local region across the entire input. Photo credit: Paula Goodfellow.

efficiency of a linear function for detecting edges in an image.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance** to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g . For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x, y) = I(x - 1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I , then apply convolution, the result will be the same as if we applied convolution to I' , then applied the transformation

g to the output. When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual’s face, we probably want to extract different features at different locations—the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

Finally, some kinds of data cannot be processed by neural networks defined by matrix multiplication with a fixed-shape matrix. Convolution enables processing of some of these kinds of data. We discuss this further in section 9.7.

9.3 Pooling

A typical layer of a convolutional network consists of three stages (see figure 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the **detector stage**. In the third stage, we use a **pooling function** to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the **max pooling** (Zhou and Chellappa, 1988) operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

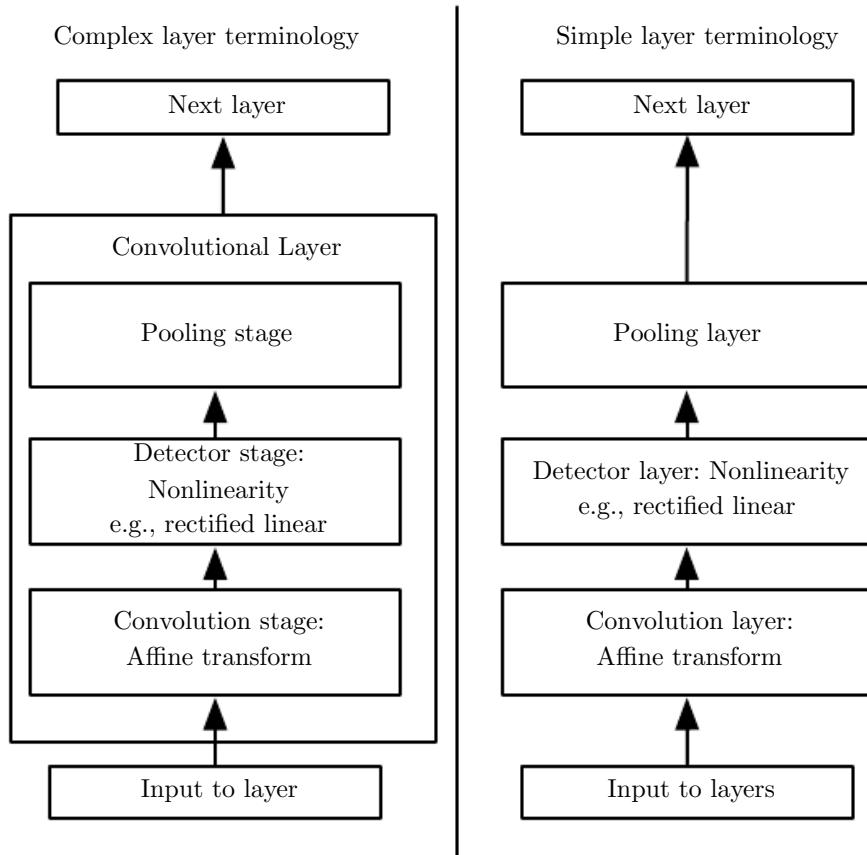


Figure 9.7: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. (*Left*) In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many “stages.” In this terminology, there is a one-to-one mapping between kernel tensors and network layers. In this book we generally use this terminology. (*Right*) In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every “layer” has parameters.

In all cases, pooling helps to make the representation approximately **invariant** to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. See figure 9.8 for an example of how this works. *Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.* For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature. For example, if we want to find a corner defined

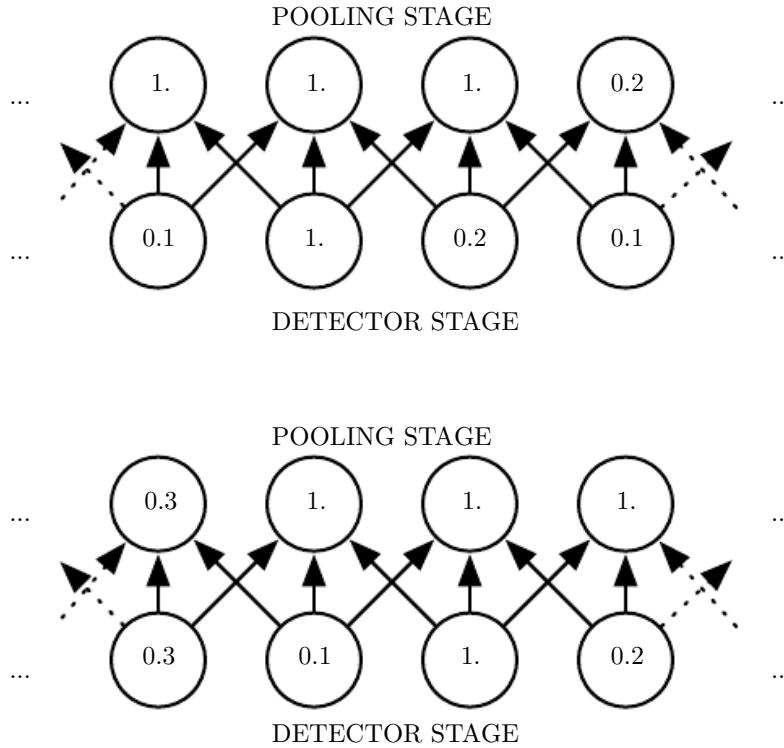


Figure 9.8: Max pooling introduces invariance. (*Top*)A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. (*Bottom*)A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are sensitive only to the maximum value in the neighborhood, not its exact location.

by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet.

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (see figure 9.9).

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k pixels apart rather than 1 pixel apart. See figure 9.10 for an example. This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When

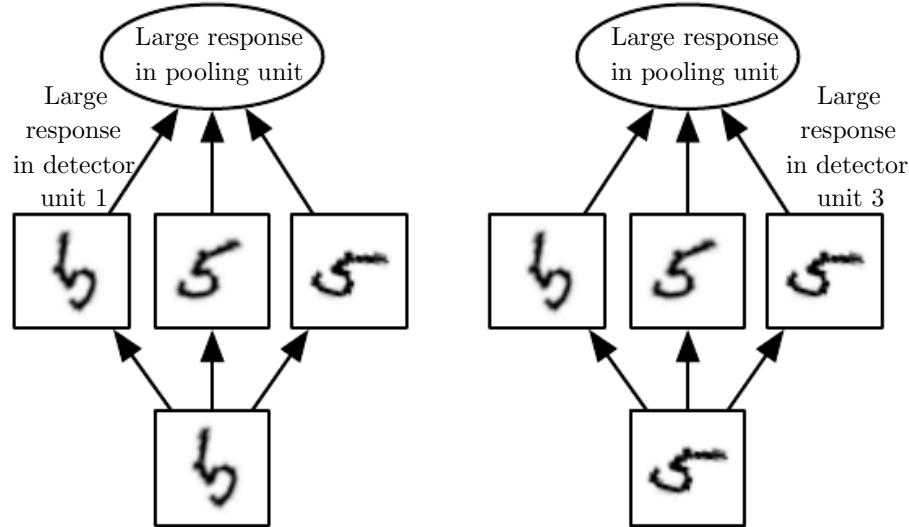


Figure 9.9: Example of learned invariances. A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand written 5. Each filter attempts to match a slightly different orientation of the 5. When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which detector unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way. This principle is leveraged by maxout networks ([Goodfellow et al., 2013a](#)) and other convolutional networks. Max pooling over spatial positions is naturally invariant to translation; this multichannel approach is only necessary for learning other transformations.

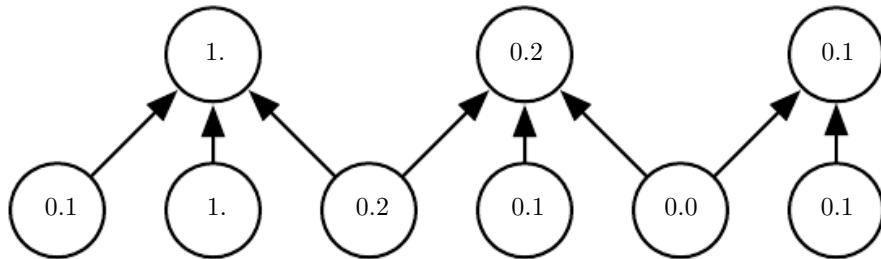


Figure 9.10: Pooling with downsampling. Here we use max pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer. Note that the rightmost pooling region has a smaller size but must be included if we do not want to ignore some of the detector units.

the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication), this

reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau *et al.*, 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau *et al.*, 2011). This approach yields a different set of pooling regions for each image. Another approach is to *learn* a single pooling structure that is then applied to all images (Jia *et al.*, 2012).

Pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders. These issues are discussed further when we present these types of networks in part III. Pooling in convolutional Boltzmann machines is presented in section 20.6. The inverse-like operations on pooling units needed in some differentiable networks are covered in section 20.10.6.

Some examples of complete convolutional network architectures for classification using convolution and pooling are shown in figure 9.11.

9.4 Convolution and Pooling as an Infinitely Strong Prior

Recall the concept of a **prior probability distribution** from section 5.6. This is a probability distribution over the parameters of a model that encodes our beliefs about what models are reasonable, before we have seen any data.

Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a prior distribution with high entropy, such as a Gaussian distribution with high variance. Such a prior allows the data to move the parameters more or less freely. A strong prior has very low entropy, such as a Gaussian distribution with low variance. Such a prior plays a more active role in determining where the parameters end up.

An infinitely strong prior places zero probability on some parameters and says

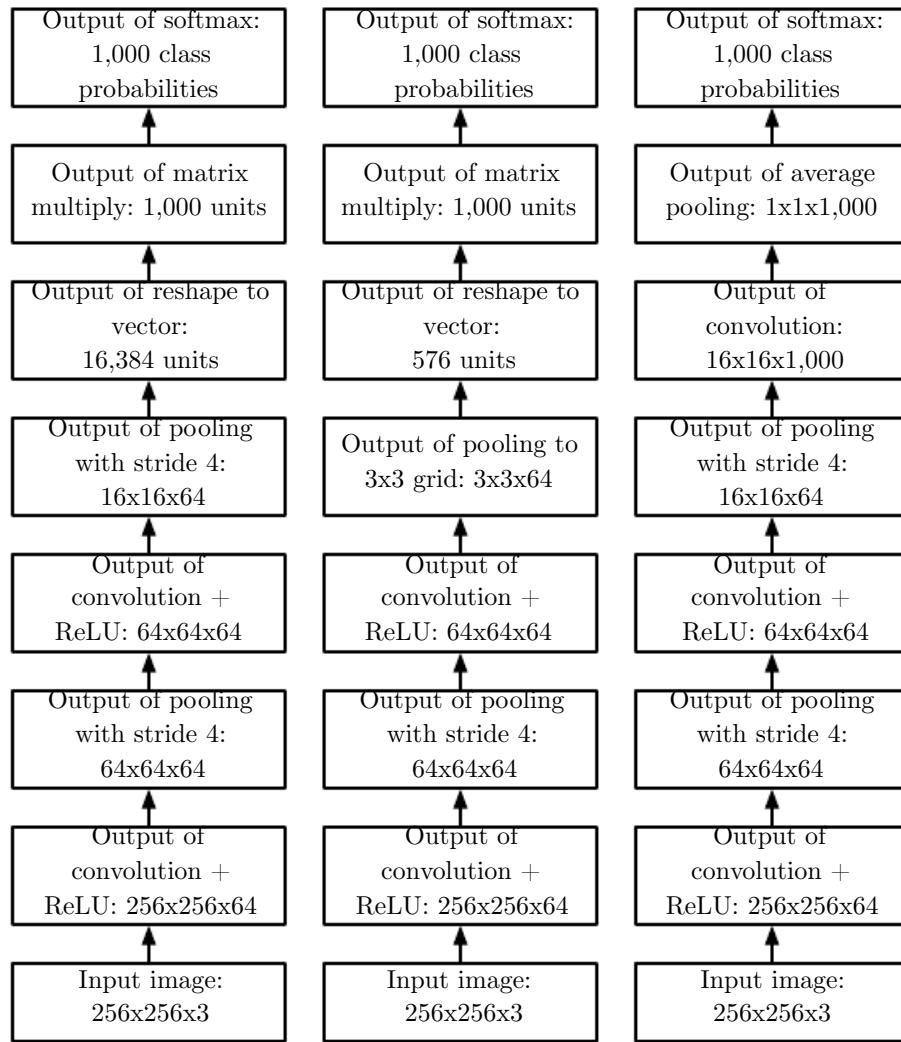


Figure 9.11: Examples of architectures for classification with convolutional networks. The specific strides and depths used in this figure are not advisable for real use; they are designed to be very shallow to fit onto the page. Real convolutional networks also often involve significant amounts of branching, unlike the chain structures used here for simplicity.

(Left) A convolutional network that processes a fixed image size. After alternating between convolution and pooling for a few layers, the tensor for the convolutional feature map is reshaped to flatten out the spatial dimensions. The rest of the network is an ordinary feedforward network classifier, as described in chapter 6.

(Center) A convolutional network that processes a variably sized image but still maintains a fully connected section. This network uses a pooling operation with variably sized pools but a fixed number of pools, in order to provide a fixed-size vector of 576 units to the fully connected portion of the network.

(Right) A convolutional network that does not have any fully connected weight layer. Instead, the last convolutional layer outputs one feature map per class. The model presumably learns a map of how likely each class is to occur at each spatial location. Averaging a feature map down to a single value provides the argument to the softmax classifier at the top.

that these parameter values are completely forbidden, regardless of how much support the data give to those values.

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit. Overall, we can think of the use of convolution as introducing an infinitely strong prior probability distribution over the parameters of a layer. This prior says that the function the layer should learn contains only local interactions and is equivariant to translation. Likewise, the use of pooling is an infinitely strong prior that each unit should be invariant to small translations.

Of course, implementing a convolutional net as a fully connected net with an infinitely strong prior would be extremely wasteful computationally. But thinking of a convolutional net as a fully connected net with an infinitely strong prior can give us some insights into how convolutional nets work.

One key insight is that convolution and pooling can cause underfitting. Like any prior, convolution and pooling are only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving precise spatial information, then using pooling on all features can increase the training error. Some convolutional network architectures ([Szegedy et al., 2014a](#)) are designed to use pooling on some channels but not on other channels, in order to get both highly invariant features and features that will not underfit when the translation invariance prior is incorrect. When a task involves incorporating information from very distant locations in the input, then the prior imposed by convolution may be inappropriate.

Another key insight from this view is that we should only compare convolutional models to other convolutional models in benchmarks of statistical learning performance. Models that do not use convolution would be able to learn even if we permuted all the pixels in the image. For many image datasets, there are separate benchmarks for models that are **permutation invariant** and must discover the concept of topology via learning and for models that have the knowledge of spatial relationships hard coded into them by their designer.

9.5 Variants of the Basic Convolution Function

When discussing convolution in the context of neural networks, we usually do not refer exactly to the standard discrete convolution operation as it is usually understood in the mathematical literature. The functions used in practice differ slightly. Here we describe these differences in detail and highlight some useful properties of the functions used in neural networks.

First, when we refer to convolution in the context of neural networks, we usually actually mean an operation that consists of many applications of convolution in parallel. This is because convolution with a single kernel can extract only one kind of feature, albeit at many spatial locations. Usually we want each layer of our network to extract many kinds of features, at many locations.

Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations. For example, a color image has a red, green and blue intensity at each pixel. In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel. Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch, but we will omit the batch axis in our description here for simplicity.

Because convolutional networks usually use multichannel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel flipping is used. These multichannel operations are only commutative if each operation has the same number of output channels as input channels.

Assume we have a 4-D kernel tensor \mathbf{K} with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit. Assume our input consists of observed data \mathbf{V} with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k . Assume our output consists of \mathbf{Z} with the same format as \mathbf{V} . If \mathbf{Z} is produced by convolving \mathbf{K} across \mathbf{V} without flipping \mathbf{K} , then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}, \quad (9.7)$$

where the summation over l , m and n is over all values for which the tensor indexing operations inside the summation are valid. In linear algebra notation,

we index into arrays using a 1 for the first entry. This necessitates the -1 in the above formula. Programming languages such as C and Python index starting from 0, rendering the above expression even simpler.

We may want to skip over some positions of the kernel to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as downsampling the output of the full convolution function. If we want to sample only every s pixels in each direction in the output, then we can define a downsampled convolution function c such that

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1)\times s+m, (k-1)\times s+n} K_{i,l,m,n}]. \quad (9.8)$$

We refer to s as the **stride** of this downsampled convolution. It is also possible to define a separate stride for each direction of motion. See figure 9.12 for an illustration.

One essential feature of any convolutional network implementation is the ability to implicitly zero pad the input \mathbf{V} to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels—both scenarios that significantly limit the expressive power of the network. See figure 9.13 for an example.

Three special cases of the zero-padding setting are worth mentioning. One is the extreme case in which no zero padding is used whatsoever, and the convolution kernel is allowed to visit only positions where the entire kernel is contained entirely within the image. In MATLAB terminology, this is called **valid** convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer. If the input image has width m and the kernel has width k , the output will be of width $m - k + 1$. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to 1×1 , at which point additional layers cannot meaningfully be considered convolutional. Another special case of the zero-padding setting is when just enough zero padding is added to keep the size of the output equal to the size of the input. MATLAB calls this **same** convolution. In this case, the network can contain as many convolutional layers as the available hardware can support, since the operation of convolution does not modify the architectural possibilities

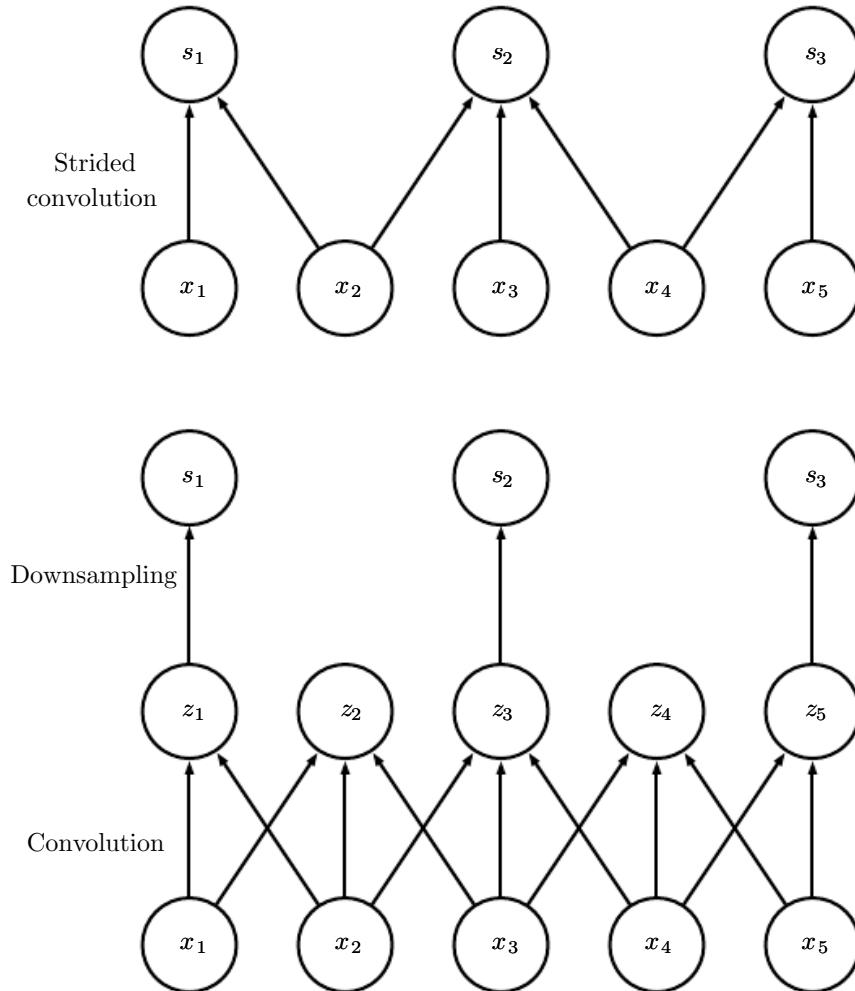


Figure 9.12: Convolution with a stride. In this example, we use a stride of two. (*Top*) Convolution with a stride length of two implemented in a single operation. (*Bottom*) Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by downsampling. Obviously, the two-step approach involving downsampling is computationally wasteful, because it computes many values that are then discarded.

available to the next layer. The input pixels near the border, however, influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model. This motivates the other extreme case, which MATLAB refers to as **full** convolution, in which enough zeros are added for every pixel to be visited k times in each direction, resulting in an output image of width $m + k - 1$. In this case, the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding (in

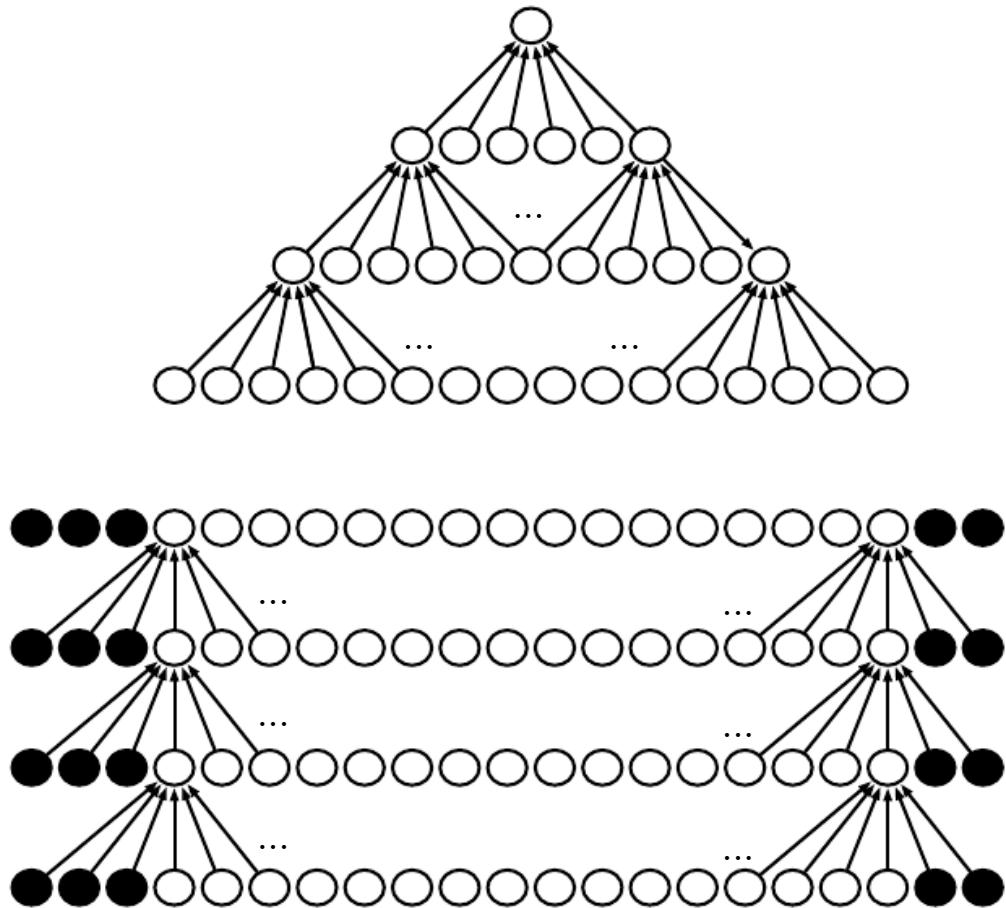


Figure 9.13: The effect of zero padding on network size. Consider a convolutional network with a kernel of width six at every layer. In this example, we do not use any pooling, so only the convolution operation itself shrinks the network size. (*Top*) In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive, and some shrinking is inevitable in this kind of architecture. (*Bottom*) By adding five implicit zeros to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

terms of test set classification accuracy) lies somewhere between “valid” and “same” convolution.

In some cases, we do not actually want to use convolution, but want to use locally connected layers instead (LeCun, 1986, 1989). In this case, the adjacency matrix in the graph of our MLP is the same, but every connection has its own weight, specified by a 6-D tensor \mathbf{W} . The indices into \mathbf{W} are respectively: i ; the

output channel; j , the output row; k , the output column; l , the input channel; m , the row offset within the input; and n , the column offset within the input. The linear part of a locally connected layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]. \quad (9.9)$$

This is sometimes also called **unshared convolution**, because it is a similar operation to discrete convolution with a small kernel, but without sharing parameters across locations. Figure 9.14 compares local connections, convolution, and full connections.

Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space. For example, if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image.

It can also be useful to make versions of convolution or locally connected layers in which the connectivity is further restricted, for example to constrain each output channel i to be a function of only a subset of the input channels l . A common way to do this is to make the first m output channels connect to only the first n input channels, the second m output channels connect to only the second n input channels, and so on. See figure 9.15 for an example. Modeling interactions between few channels allows the network to have fewer parameters, reducing memory consumption, increasing statistical efficiency, and reducing the amount of computation needed to perform forward and back-propagation. It accomplishes these goals without reducing the number of hidden units.

Tiled convolution (Gregor and LeCun, 2010a; Le *et al.*, 2010) offers a compromise between a convolutional layer and a locally connected layer. Rather than learning a separate set of weights at *every* spatial location, we learn a set of kernels that we rotate through as we move through space. This means that immediately neighboring locations will have different filters, as in a locally connected layer, but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels, rather than by the size of the entire output feature map. See figure 9.16 for a comparison of locally connected layers, tiled convolution, and standard convolution.

To define tiled convolution algebraically, let \mathbf{K} be a 6-D tensor, where two of the dimensions correspond to different locations in the output map. Rather than having a separate index for each location in the output map, output locations cycle through a set of t different choices of kernel stack in each direction. If t is equal to

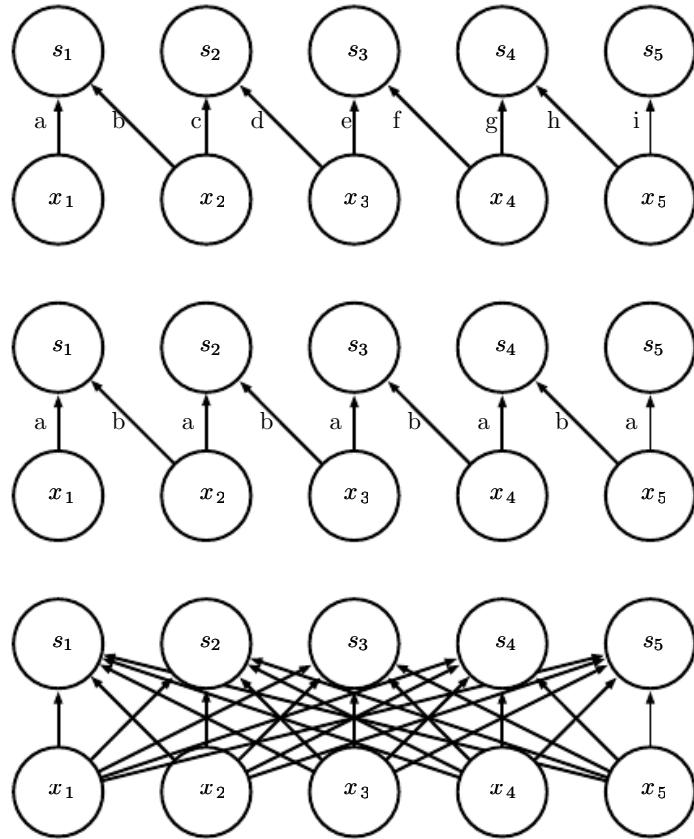


Figure 9.14: Comparison of local connections, convolution, and full connections.

(Top) A locally connected layer with a patch size of two pixels. Each edge is labeled with a unique letter to show that each edge is associated with its own weight parameter.

(Center) A convolutional layer with a kernel width of two pixels. This model has exactly the same connectivity as the locally connected layer. The difference lies not in which units interact with each other, but in how the parameters are shared. The locally connected layer has no parameter sharing. The convolutional layer uses the same two weights repeatedly across the entire input, as indicated by the repetition of the letters labeling each edge.

(Bottom) A fully connected layer resembles a locally connected layer in the sense that each edge has its own parameter (there are too many to label explicitly with letters in this diagram). It does not, however, have the restricted connectivity of the locally connected layer.

the output width, this is the same as a locally connected layer.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j \% t+1, k \% t+1}, \quad (9.10)$$

where percent is the modulo operation, with $t \% t = 0$, $(t + 1) \% t = 1$, and so on. It is straightforward to generalize this equation to use a different tiling range for each dimension.

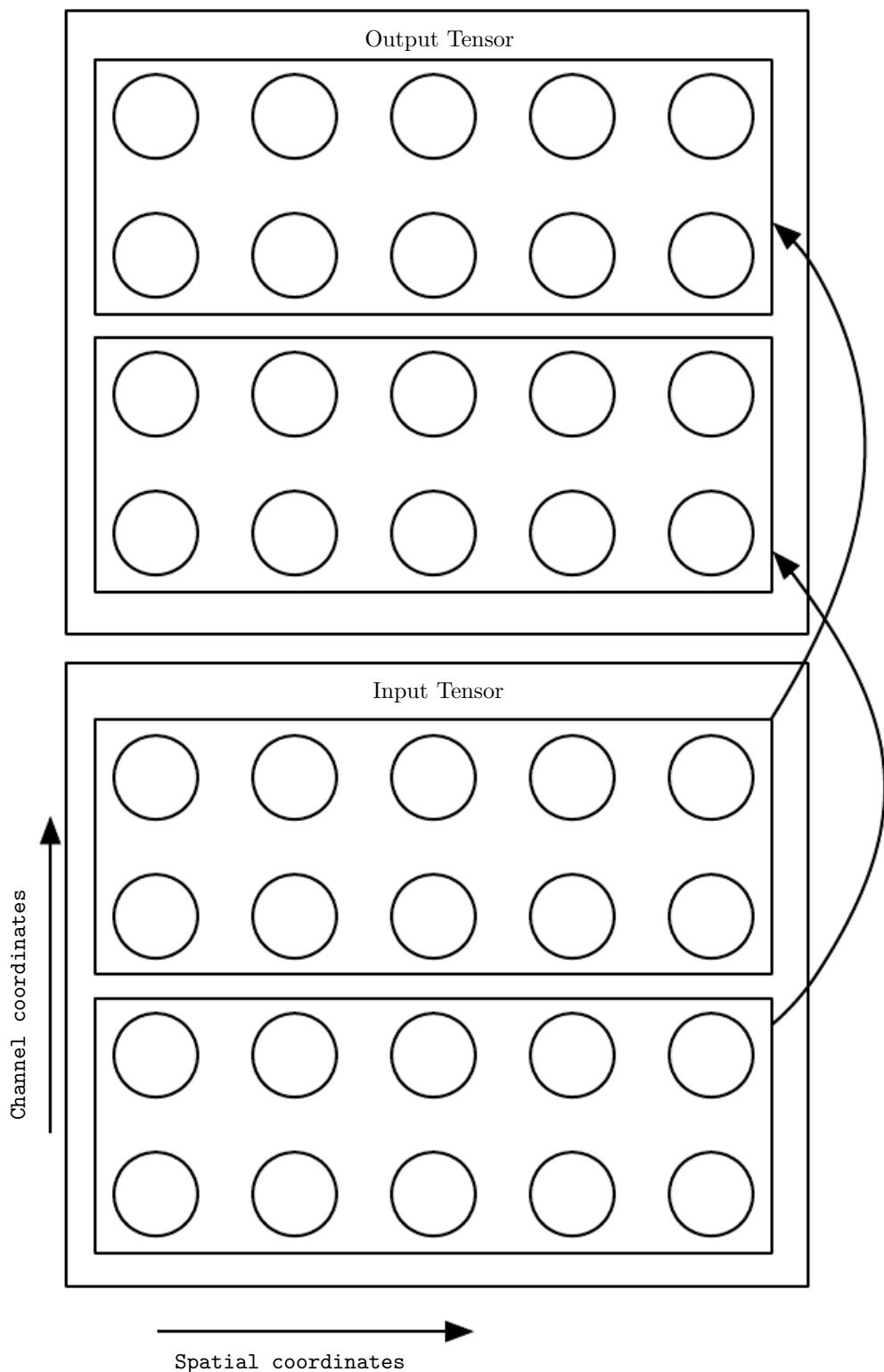


Figure 9.15: A convolutional network with the first two output channels connected to only the first two input channels, and the second two output channels connected to only the second two input channels.

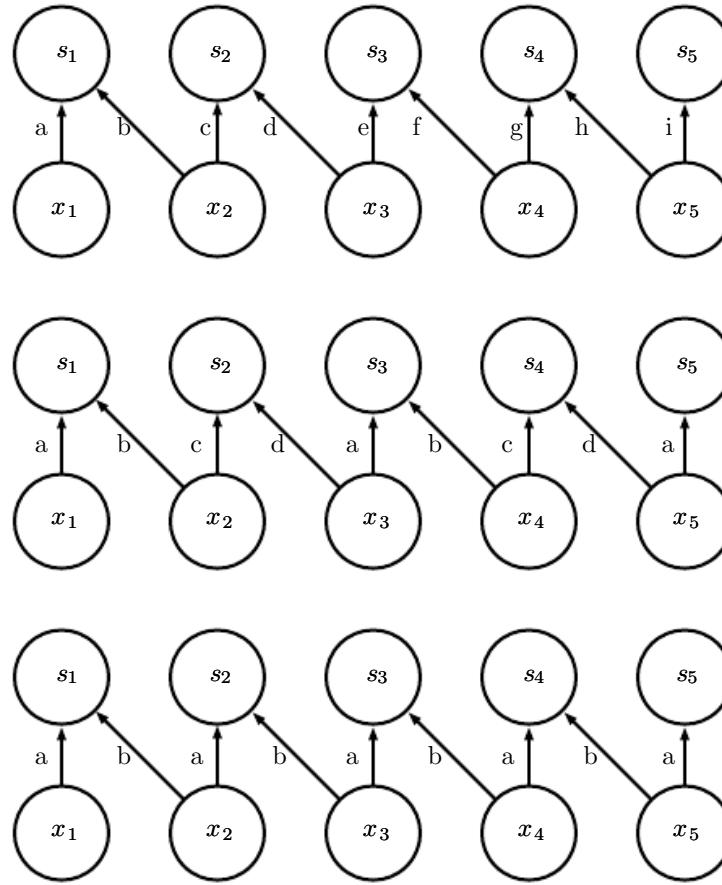


Figure 9.16: A comparison of locally connected layers, tiled convolution, and standard convolution. All three have the same sets of connections between units, when the same size of kernel is used. This diagram illustrates the use of a kernel that is two pixels wide. The differences between the methods lies in how they share parameters. (*Top*) A locally connected layer has no sharing at all. We indicate that each connection has its own weight by labeling each connection with a unique letter. (*Center*) Tiled convolution has a set of t different kernels. Here we illustrate the case of $t = 2$. One of these kernels has edges labeled “a” and “b,” while the other has edges labeled “c” and “d.” Each time we move one pixel to the right in the output, we move on to using a different kernel. This means that, like the locally connected layer, neighboring units in the output have different parameters. Unlike the locally connected layer, after we have gone through all t available kernels, we cycle back to the first kernel. If two output units are separated by a multiple of t steps, then they share parameters. (*Bottom*) Traditional convolution is equivalent to tiled convolution with $t = 1$. There is only one kernel, and it is applied everywhere, as indicated in the diagram by using the kernel with weights labeled “a” and “b” everywhere.

Locally connected layers and tiled convolutional layers both have an interesting interaction with max pooling: the detector units of these layers are driven by different filters. If these filters learn to detect different transformed versions of

the same underlying features, then the max-pooled units become invariant to the learned transformation (see figure 9.9). Convolutional layers are hard coded to be invariant specifically to translation.

Other operations besides convolution are usually necessary to implement a convolutional network. To perform learning, one must be able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs. In some simple cases, this operation can be performed using the convolution operation, but many cases of interest, including the case of stride greater than 1, do not have this property.

Recall that convolution is a linear operation and can thus be described as a matrix multiplication (if we first reshape the input tensor into a flat vector). The matrix involved is a function of the convolution kernel. The matrix is sparse, and each element of the kernel is copied to several elements of the matrix. This view helps us to derive some of the other operations needed to implement a convolutional network.

Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to back-propagate error derivatives through a convolutional layer, so it is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed if we wish to reconstruct the visible units from the hidden units (Simard *et al.*, 1992). Reconstructing the visible units is an operation commonly used in the models described in part III of this book, such as autoencoders, RBMs, and sparse coding.

Transpose convolution is necessary to construct convolutional versions of those models. Like the kernel gradient operation, this input gradient operation can sometimes be implemented using a convolution but in general requires a third operation to be implemented. Care must be taken to coordinate this transpose operation with the forward propagation. The size of the output that the transpose operation should return depends on the zero-padding policy and stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

These three operations—convolution, backprop from output to weights, and backprop from output to inputs—are sufficient to compute all the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution. See Goodfellow (2010) for a full derivation of the equations in the fully general multidimensional, multieexample case. To give a sense of how these

equations work, we present the two-dimensional, single example version here.

Suppose we want to train a convolutional network that incorporates strided convolution of kernel stack \mathbf{K} applied to multichannel image \mathbf{V} with stride s as defined by $c(\mathbf{K}, \mathbf{V}, s)$, as in equation 9.8. Suppose we want to minimize some loss function $J(\mathbf{V}, \mathbf{K})$. During forward propagation, we will need to use c itself to output \mathbf{Z} , which is then propagated through the rest of the network and used to compute the cost function J . During back-propagation, we will receive a tensor \mathbf{G} such that $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$.

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k, (n-1)\times s+l}. \quad (9.11)$$

If this layer is not the bottom layer of the network, we will need to compute the gradient with respect to \mathbf{V} to back-propagate the error further down. To do so, we can use a function

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) \quad (9.12)$$

$$= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)\times s+p=k}} \sum_q K_{q,i,m,p} G_{q,l,n}. \quad (9.13)$$

Autoencoder networks, described in chapter 14, are feedforward networks trained to copy their input to their output. A simple example is the PCA algorithm, which copies its input \mathbf{x} to an approximate reconstruction \mathbf{r} using the function $\mathbf{W}^\top \mathbf{W} \mathbf{x}$. It is common for more general autoencoders to use multiplication by the transpose of the weight matrix just as PCA does. To make such models convolutional, we can use the function h to perform the transpose of the convolution operation. Suppose we have hidden units \mathbf{H} in the same format as \mathbf{Z} and we define a reconstruction

$$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s). \quad (9.14)$$

To train the autoencoder, we will receive the gradient with respect to \mathbf{R} as a tensor \mathbf{E} . To train the decoder, we need to obtain the gradient with respect to \mathbf{K} . This is given by $g(\mathbf{H}, \mathbf{E}, s)$. To train the encoder, we need to obtain the gradient with respect to \mathbf{H} . This is given by $c(\mathbf{K}, \mathbf{E}, s)$. It is also possible to differentiate through g using c and h , but these operations are not needed for the back-propagation algorithm on any standard network architectures.

Generally, we do not use only a linear operation to transform from the inputs to the outputs in a convolutional layer. We generally also add some bias term to each output before applying the nonlinearity. This raises the question of how to share parameters among the biases. For locally connected layers, it is natural to give each unit its own bias, and for tiled convolution, it is natural to share the biases with the same tiling pattern as the kernels. For convolutional layers, it is typical to have one bias per channel of the output and share it across all locations within each convolution map. If the input is of known, fixed size, however, it is also possible to learn a separate bias at each location of the output map. Separating the biases may slightly reduce the statistical efficiency of the model, but it allows the model to correct for differences in the image statistics at different locations. For example, when using implicit zero padding, detector units at the edge of the image receive less total input and may need larger biases.

9.6 Structured Outputs

Convolutional networks can be used to output a high-dimensional structured object, rather than just predicting a class label for a classification task or a real value for a regression task. Typically this object is just a tensor, emitted by a standard convolutional layer. For example, the model might emit a tensor \mathbf{S} , where $S_{i,j,k}$ is the probability that pixel (j, k) of the input to the network belongs to class i . This allows the model to label every pixel in an image and draw precise masks that follow the outlines of individual objects.

One issue that often comes up is that the output plane can be smaller than the input plane, as shown in figure 9.13. In the kinds of architectures typically used for classification of a single object in an image, the greatest reduction in the spatial dimensions of the network comes from using pooling layers with large stride. To produce an output map of similar size as the input, one can avoid pooling altogether (Jain *et al.*, 2007). Another strategy is to simply emit a lower-resolution grid of labels (Pinheiro and Collobert, 2014, 2015). Finally, in principle, one could use a pooling operator with unit stride.

One strategy for pixel-wise labeling of images is to produce an initial guess of the image labels, then refine this initial guess using the interactions between neighboring pixels. Repeating this refinement step several times corresponds to using the same convolutions at each stage, sharing weights between the last layers of the deep net (Jain *et al.*, 2007). This makes the sequence of computations performed by the successive convolutional layers with weights shared across layers a particular

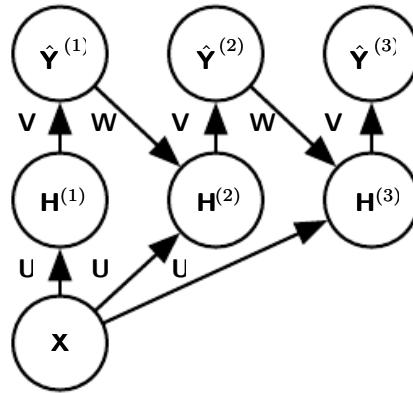


Figure 9.17: An example of a recurrent convolutional network for pixel labeling. The input is an image tensor \mathbf{X} , with axes corresponding to image rows, image columns, and channels (red, green, blue). The goal is to output a tensor of labels \hat{Y} , with a probability distribution over labels for each pixel. This tensor has axes corresponding to image rows, image columns, and the different classes. Rather than outputting \hat{Y} in a single shot, the recurrent network iteratively refines its estimate \hat{Y} by using a previous estimate of \hat{Y} as input for creating a new estimate. The same parameters are used for each updated estimate, and the estimate can be refined as many times as we wish. The tensor of convolution kernels \mathbf{U} is used on each step to compute the hidden representation given the input image. The kernel tensor \mathbf{V} is used to produce an estimate of the labels given the hidden values. On all but the first step, the kernels \mathbf{W} are convolved over \hat{Y} to provide input to the hidden layer. On the first time step, this term is replaced by zero. Because the same parameters are used on each step, this is an example of a recurrent network, as described in chapter 10.

kind of recurrent network (Pinheiro and Collobert, 2014, 2015). Figure 9.17 shows the architecture of such a recurrent convolutional network.

Once a prediction for each pixel is made, various methods can be used to further process these predictions to obtain a segmentation of the image into regions (Briggman *et al.*, 2009; Turaga *et al.*, 2010; Farabet *et al.*, 2013). The general idea is to assume that large groups of contiguous pixels tend to be associated with the same label. Graphical models can describe the probabilistic relationships between neighboring pixels. Alternatively, the convolutional network can be trained to maximize an approximation of the graphical model training objective (Ning *et al.*, 2005; Thompson *et al.*, 2014).

9.7 Data Types

The data used with a convolutional network usually consist of several channels, each channel being the observation of a different quantity at some point in space or time. See table 9.1 for examples of data types with different dimensionalities and number of channels.

For an example of convolutional networks applied to video, see [Chen et al. \(2010\)](#).

So far we have discussed only the case where every example in the train and test data has the same spatial dimensions. One advantage to convolutional networks is that they can also process inputs with varying spatial extents. These kinds of input simply cannot be represented by traditional, matrix multiplication-based neural networks. This provides a compelling reason to use convolutional networks even when computational cost and overfitting are not significant issues.

For example, consider a collection of images in which each image has a different width and height. It is unclear how to model such inputs with a weight matrix of fixed size. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly. Convolution may be viewed as matrix multiplication; the same convolution kernel induces a different size of doubly block circulant matrix for each size of input. Sometimes the output of the network as well as the input is allowed to have variable size, for example, if we want to assign a class label to each pixel of the input. In this case, no further design work is necessary. In other cases, the network must produce some fixed-size output, for example, if we want to assign a single class label to the entire image. In this case, we must make some additional design steps, like inserting a pooling layer whose pooling regions scale in size proportional to the size of the input, to maintain a fixed number of pooled outputs. Some examples of this kind of strategy are shown in figure 9.11.

Note that the use of convolution for processing variably sized inputs makes sense only for inputs that have variable size because they contain varying amounts of observation of the same kind of thing—different lengths of recordings over time, different widths of observations over space, and so forth. Convolution does not make sense if the input has variable size because it can optionally include different kinds of observations. For example, if we are processing college applications, and our features consist of both grades and standardized test scores, but not every applicant took the standardized test, then it does not make sense to convolve the same weights over features corresponding to the grades as well as the features

	Single channel	Multichannel
1-D	Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step.	Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a “skeleton” over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character’s skeleton. Each channel in the data we feed to the convolutional model represents the angle about one axis of one joint.
2-D	Audio data that has been preprocessed with a Fourier transform: We can transform the audio waveform into a 2-D tensor with different rows corresponding to different frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time. Using convolution across the frequency axis makes the model equivariant to frequency, so that the same melody played in a different octave produces the same representation but at a different height in the network’s output.	Color image data: One channel contains the red pixels, one the green pixels, and one the blue pixels. The convolution kernel moves over both the horizontal and the vertical axes of the image, conferring translation equivariance in both directions.
3-D	Volumetric data: A common source of this kind of data is medical imaging technology, such as CT scans.	Color video data: One axis corresponds to time, one to the height of the video frame, and one to the width of the video frame.

Table 9.1: Examples of different formats of data that can be used with convolutional networks.

corresponding to the test scores.

9.8 Efficient Convolution Algorithms

Modern convolutional network applications often involve networks containing more than one million units. Powerful implementations exploiting parallel computation resources, as discussed in section 12.1, are essential. In many cases, however, it is also possible to speed up convolution by selecting an appropriate convolution algorithm.

Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform. For some problem sizes, this can be faster than the naive implementation of discrete convolution.

When a d -dimensional kernel can be expressed as the outer product of d vectors, one vector per dimension, the kernel is called **separable**. When the kernel is separable, naive convolution is inefficient. It is equivalent to compose d one-dimensional convolutions with each of these vectors. The composed approach is significantly faster than performing one d -dimensional convolution with their outer product. The kernel also takes fewer parameters to represent as vectors. If the kernel is w elements wide in each dimension, then naive multidimensional convolution requires $O(w^d)$ runtime and parameter storage space, while separable convolution requires $O(w \times d)$ runtime and parameter storage space. Of course, not every convolution can be represented in this way.

Devising faster ways of performing convolution or approximate convolution without harming the accuracy of the model is an active area of research. Even techniques that improve the efficiency of only forward propagation are useful because in the commercial setting, it is typical to devote more resources to deployment of a network than to its training.

9.9 Random or Unsupervised Features

Typically, the most expensive part of convolutional network training is learning the features. The output layer is usually relatively inexpensive because of the small number of features provided as input to this layer after passing through several layers of pooling. When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and

backward propagation through the entire network. One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.

There are three basic strategies for obtaining convolution kernels without supervised training. One is to simply initialize them randomly. Another is to design them by hand, for example, by setting each kernel to detect edges at a certain orientation or scale. Finally, one can learn the kernels with an unsupervised criterion. For example, Coates *et al.* (2011) apply k -means clustering to small image patches, then use each learned centroid as a convolution kernel. In Part III we describe many more unsupervised learning approaches. Learning the features with an unsupervised criterion allows them to be determined separately from the classifier layer at the top of the architecture. One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer. Learning the last layer is then typically a convex optimization problem, assuming the last layer is something like logistic regression or an SVM.

Random filters often work surprisingly well in convolutional networks (Jarrett *et al.*, 2009; Saxe *et al.*, 2011; Pinto *et al.*, 2011; Cox and Pinto, 2011). Saxe *et al.* (2011) showed that layers consisting of convolution followed by pooling naturally become frequency selective and translation invariant when assigned random weights. They argue that this provides an inexpensive way to choose the architecture of a convolutional network: first, evaluate the performance of several convolutional network architectures by training only the last layer; then take the best of these architectures and train the entire architecture using a more expensive approach.

An intermediate approach is to learn the features, but using methods that do not require full forward and back-propagation at every gradient step. As with multilayer perceptrons, we use greedy layer-wise pretraining, to train the first layer in isolation, then extract all features from the first layer only once, then train the second layer in isolation given those features, and so on. In chapter 8 we described how to perform supervised greedy layer-wise pretraining, and in part III extend this to greedy layer-wise pretraining using an unsupervised criterion at each layer. The canonical example of greedy layer-wise pretraining of a convolutional model is the convolutional deep belief network (Lee *et al.*, 2009). Convolutional networks offer us the opportunity to take the pretraining strategy one step further than is possible with multilayer perceptrons. Instead of training an entire convolutional layer at a time, we can train a model of a small patch, as Coates *et al.* (2011) do with k -means. We can then use the parameters from this patch-based model to define the kernels of a convolutional layer. This means that it is possible to use unsupervised learning to train a convolutional network *without ever using convolution during the training*

process. Using this approach, we can train very large models and incur a high computational cost only at inference time (Ranzato *et al.*, 2007b; Jarrett *et al.*, 2009; Kavukcuoglu *et al.*, 2010; Coates *et al.*, 2013). This approach was popular from roughly 2007 to 2013, when labeled datasets were small and computational power was more limited. Today, most convolutional networks are trained in a purely supervised fashion, using full forward and back-propagation through the entire network on each training iteration.

As with other approaches to unsupervised pretraining, it remains difficult to tease apart the cause of some of the benefits seen with this approach. Unsupervised pretraining may offer some regularization relative to supervised training, or it may simply allow us to train much larger architectures because of the reduced computational cost of the learning rule.

9.10 The Neuroscientific Basis for Convolutional Networks

Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence. Though convolutional networks have been guided by many other fields, some of the key design principles of neural networks were drawn from neuroscience.

The history of convolutional networks begins with neuroscientific experiments long before the relevant computational models were developed. Neurophysiologists David Hubel and Torsten Wiesel collaborated for several years to determine many of the most basic facts about how the mammalian vision system works (Hubel and Wiesel, 1959, 1962, 1968). Their accomplishments were eventually recognized with a Nobel prize. Their findings that have had the greatest influence on contemporary deep learning models were based on recording the activity of individual neurons in cats. They observed how neurons in the cat’s brain responded to images projected in precise locations on a screen in front of the cat. Their great discovery was that neurons in the early visual system responded most strongly to very specific patterns of light, such as precisely oriented bars, but responded hardly at all to other patterns.

Their work helped to characterize many aspects of brain function that are beyond the scope of this book. From the point of view of deep learning, we can focus on a simplified, cartoon view of brain function.

In this simplified view, we focus on a part of the brain called V1, also known as the **primary visual cortex**. V1 is the first area of the brain that begins to

perform significantly advanced processing of visual input. In this cartoon view, images are formed by light arriving in the eye and stimulating the retina, the light-sensitive tissue in the back of the eye. The neurons in the retina perform some simple preprocessing of the image but do not substantially alter the way it is represented. The image then passes through the optic nerve and a brain region called the *lateral geniculate nucleus*. The main role, as far as we are concerned here, of both anatomical regions is primarily just to carry the signal from the eye to V1, which is located at the back of the head.

A convolutional network layer is designed to capture three properties of V1:

1. V1 is arranged in a spatial map. It actually has a two-dimensional structure, mirroring the structure of the image in the retina. For example, light arriving at the lower half of the retina affects only the corresponding half of V1. Convolutional networks capture this property by having their features defined in terms of two-dimensional maps.
2. V1 contains many **simple cells**. A simple cell’s activity can to some extent be characterized by a linear function of the image in a small, spatially localized receptive field. The detector units of a convolutional network are designed to emulate these properties of simple cells.
3. V1 also contains many **complex cells**. These cells respond to features that are similar to those detected by simple cells, but complex cells are invariant to small shifts in the position of the feature. This inspires the pooling units of convolutional networks. Complex cells are also invariant to some changes in lighting that cannot be captured simply by pooling over spatial locations. These invariances have inspired some of the cross-channel pooling strategies in convolutional networks, such as maxout units ([Goodfellow et al., 2013a](#)).

Though we know the most about V1, it is generally believed that the same basic principles apply to other areas of the visual system. In our cartoon view of the visual system, the basic strategy of detection followed by pooling is repeatedly applied as we move deeper into the brain. As we pass through multiple anatomical layers of the brain, we eventually find cells that respond to some specific concept and are invariant to many transformations of the input. These cells have been nicknamed “grandmother cells”—the idea is that a person could have a neuron that activates when seeing an image of their grandmother, regardless of whether she appears in the left or right side of the image, whether the image is a close-up of her face or zoomed-out shot of her entire body, whether she is brightly lit or in shadow, and so on.

These grandmother cells have been shown to actually exist in the human brain, in a region called the *medial temporal lobe* (Quiroga *et al.*, 2005). Researchers tested whether individual neurons would respond to photos of famous individuals. They found what has come to be called the “Halle Berry neuron,” an individual neuron that is activated by the concept of Halle Berry. This neuron fires when a person sees a photo of Halle Berry, a drawing of Halle Berry, or even text containing the words “Halle Berry.” Of course, this has nothing to do with Halle Berry herself; other neurons responded to the presence of Bill Clinton, Jennifer Aniston, and so forth.

These medial temporal lobe neurons are somewhat more general than modern convolutional networks, which would not automatically generalize to identifying a person or object when reading its name. The closest analog to a convolutional network’s last layer of features is a brain area called the *inferotemporal cortex* (IT). When viewing an object, information flows from the retina, through the LGN, to V1, then onward to V2, then V4, then IT. This happens within the first 100ms of glimpsing an object. If a person is allowed to continue looking at the object for more time, then information will begin to flow backward as the brain uses top-down feedback to update the activations in the lower level brain areas. If we interrupt the person’s gaze, however, and observe only the firing rates that result from the first 100ms of mostly feedforward activation, then IT proves to be similar to a convolutional network. Convolutional networks can predict IT firing rates and perform similarly to (time-limited) humans on object recognition tasks (DiCarlo, 2013).

That being said, there are many differences between convolutional networks and the mammalian vision system. Some of these differences are well known to computational neuroscientists but outside the scope of this book. Some of these differences are not yet known, because many basic questions about how the mammalian vision system works remain unanswered. As a brief list:

- The human eye is mostly very low resolution, except for a tiny patch called the **fovea**. The fovea only observes an area about the size of a thumbnail held at arms length. Though we feel as if we can see an entire scene in high resolution, this is an illusion created by the subconscious part of our brain, as it stitches together several glimpses of small areas. Most convolutional networks actually receive large full-resolution photographs as input. The human brain makes several eye movements called **saccades** to glimpse the most visually salient or task-relevant parts of a scene. Incorporating similar attention mechanisms into deep learning models is an active research direction. In the context of deep learning, attention mechanisms have been most successful for natural

language processing, as described in section 12.4.5.1. Several visual models with foveation mechanisms have been developed but so far have not become the dominant approach (Larochelle and Hinton, 2010; Denil *et al.*, 2012).

- The human visual system is integrated with many other senses, such as hearing, and factors like our moods and thoughts. Convolutional networks so far are purely visual.
- The human visual system does much more than just recognize objects. It is able to understand entire scenes, including many objects and relationships between objects, and it processes rich 3-D geometric information needed for our bodies to interface with the world. Convolutional networks have been applied to some of these problems, but these applications are in their infancy.
- Even simple brain areas like V1 are heavily affected by feedback from higher levels. Feedback has been explored extensively in neural network models but has not yet been shown to offer a compelling improvement.
- While feedforward IT firing rates capture much of the same information as convolutional network features, it is not clear how similar the intermediate computations are. The brain probably uses very different activation and pooling functions. An individual neuron’s activation probably is not well characterized by a single linear filter response. A recent model of V1 involves multiple quadratic filters for each neuron (Rust *et al.*, 2005). Indeed our cartoon picture of “simple cells” and “complex cells” might create a nonexistent distinction; simple cells and complex cells might both be the same kind of cell but with their “parameters” enabling a continuum of behaviors ranging from what we call “simple” to what we call “complex.”

It is also worth mentioning that neuroscience has told us relatively little about how to *train* convolutional networks. Model structures with parameter sharing across multiple spatial locations date back to early connectionist models of vision (Marr and Poggio, 1976), but these models did not use the modern back-propagation algorithm and gradient descent. For example, the neocognitron (Fukushima, 1980) incorporated most of the model architecture design elements of the modern convolutional network but relied on a layer-wise unsupervised clustering algorithm.

Lang and Hinton (1988) introduced the use of back-propagation to train **time-delay neural networks** (TDNNs). To use contemporary terminology, TDNNs are one-dimensional convolutional networks applied to time series. Back-propagation applied to these models was not inspired by any neuroscientific observa-

tion and is considered by some to be biologically implausible. Following the success of back-propagation-based training of TDNNs, LeCun *et al.* (1989) developed the modern convolutional network by applying the same training algorithm to 2-D convolution applied to images.

So far we have described how simple cells are roughly linear and selective for certain features, complex cells are more nonlinear and become invariant to some transformations of these simple cell features, and stacks of layers that alternate between selectivity and invariance can yield grandmother cells for specific phenomena. We have not yet described precisely what these individual cells detect. In a deep nonlinear network, it can be difficult to understand the function of individual cells. Simple cells in the first layer are easier to analyze, because their responses are driven by a linear function. In an artificial neural network, we can just display an image of the convolution kernel to see what the corresponding channel of a convolutional layer responds to. In a biological neural network, we do not have access to the weights themselves. Instead, we put an electrode in the neuron, display several samples of white noise images in front of the animal's retina, and record how each of these samples causes the neuron to activate. We can then fit a linear model to these responses to obtain an approximation of the neuron's weights. This approach is known as **reverse correlation** (Ringach and Shapley, 2004).

Reverse correlation shows us that most V1 cells have weights that are described by **Gabor functions**. The Gabor function describes the weight at a 2-D point in the image. We can think of an image as being a function of 2-D coordinates, $I(x, y)$. Likewise, we can think of a simple cell as sampling the image at a set of locations, defined by a set of x coordinates \mathbb{X} and a set of y coordinates \mathbb{Y} , then applying weights that are also a function of the location, $w(x, y)$. From this point of view, the response of a simple cell to an image is given by

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x, y) I(x, y). \quad (9.15)$$

Specifically, $w(x, y)$ takes the form of a Gabor function:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp(-\beta_x x'^2 - \beta_y y'^2) \cos(f x' + \phi), \quad (9.16)$$

where

$$x' = (x - x_0) \cos(\tau) + (y - y_0) \sin(\tau) \quad (9.17)$$

and

$$y' = -(x - x_0) \sin(\tau) + (y - y_0) \cos(\tau). \quad (9.18)$$

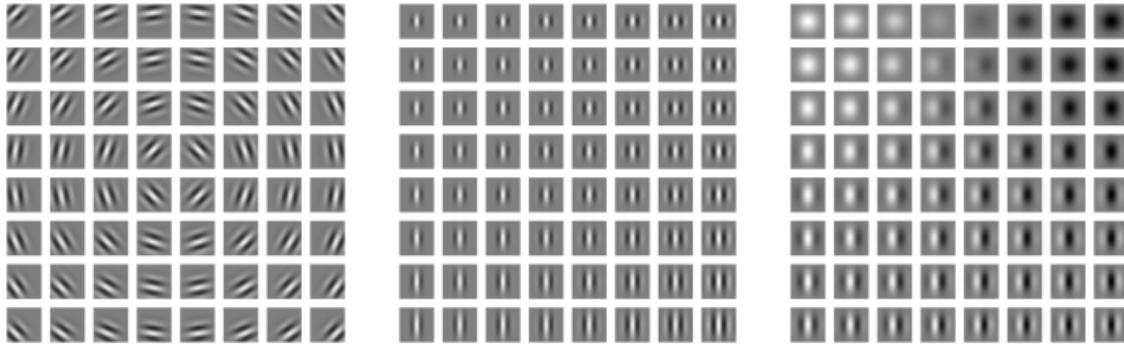


Figure 9.18: Gabor functions with a variety of parameter settings. White indicates large positive weight, black indicates large negative weight, and the background gray corresponds to zero weight. (*Left*)Gabor functions with different values of the parameters that control the coordinate system: x_0 , y_0 , and τ . Each Gabor function in this grid is assigned a value of x_0 and y_0 proportional to its position in its grid, and τ is chosen so that each Gabor filter is sensitive to the direction radiating out from the center of the grid. For the other two plots, x_0 , y_0 , and τ are fixed to zero. (*Center*)Gabor functions with different Gaussian scale parameters β_x and β_y . Gabor functions are arranged in increasing width (decreasing β_x) as we move left to right through the grid, and increasing height (decreasing β_y) as we move top to bottom. For the other two plots, the β values are fixed to 1.5 times the image width. (*Right*)Gabor functions with different sinusoid parameters f and ϕ . As we move top to bottom, f increases, and as we move left to right, ϕ increases. For the other two plots, ϕ is fixed to 0 and f is fixed to 5 times the image width.

Here, α , β_x , β_y , f , ϕ , x_0 , y_0 , and τ are parameters that control the properties of the Gabor function. Figure 9.18 shows some examples of Gabor functions with different settings of these parameters.

The parameters x_0 , y_0 , and τ define a coordinate system. We translate and rotate x and y to form x' and y' . Specifically, the simple cell will respond to image features centered at the point (x_0, y_0) , and it will respond to changes in brightness as we move along a line rotated τ radians from the horizontal.

Viewed as a function of x' and y' , the function w then responds to changes in brightness as we move along the x' axis. It has two important factors: one is a Gaussian function, and the other is a cosine function.

The Gaussian factor $\alpha \exp(-\beta_x x'^2 - \beta_y y'^2)$ can be seen as a gating term that ensures that the simple cell will respond only to values near where x' and y' are both zero, in other words, near the center of the cell's receptive field. The scaling factor α adjusts the total magnitude of the simple cell's response, while β_x and β_y control how quickly its receptive field falls off.

The cosine factor $\cos(fx' + \phi)$ controls how the simple cell responds to changing brightness along the x' axis. The parameter f controls the frequency of the cosine, and ϕ controls its phase offset.

Altogether, this cartoon view of simple cells means that a simple cell responds to a specific spatial frequency of brightness in a specific direction at a specific location. Simple cells are most excited when the wave of brightness in the image has the same phase as the weights. This occurs when the image is bright where the weights are positive and dark where the weights are negative. Simple cells are most inhibited when the wave of brightness is fully out of phase with the weights—when the image is dark where the weights are positive and bright where the weights are negative.

The cartoon view of a complex cell is that it computes the L^2 norm of the 2-D vector containing two simple cells' responses: $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. An important special case occurs when s_1 has all the same parameters as s_0 except for ϕ , and ϕ is set such that s_1 is one quarter cycle out of phase with s_0 . In this case, s_0 and s_1 form a **quadrature pair**. A complex cell defined in this way responds when the Gaussian reweighted image $I(x, y) \exp(-\beta_x x'^2 - \beta_y y'^2)$ contains a high-amplitude sinusoidal wave with frequency f in direction τ near (x_0, y_0) , *regardless of the phase offset of this wave*. In other words, the complex cell is invariant to small translations of the image in direction τ , or to negating the image (replacing black with white and vice versa).

Some of the most striking correspondences between neuroscience and machine learning come from visually comparing the features learned by machine learning models with those employed by V1. [Olshausen and Field \(1996\)](#) showed that a simple unsupervised learning algorithm, sparse coding, learns features with receptive fields similar to those of simple cells. Since then, we have found that an extremely wide variety of statistical learning algorithms learn features with Gabor-like functions when applied to natural images. This includes most deep learning algorithms, which learn these features in their first layer. Figure 9.19 shows some examples. Because so many different learning algorithms learn edge detectors, it is difficult to conclude that any specific learning algorithm is the “right” model of the brain just based on the features it learns (though it can certainly be a bad sign if an algorithm does *not* learn some sort of edge detector when applied to natural images). These features are an important part of the statistical structure of natural images and can be recovered by many different approaches to statistical modeling. See [Hyvärinen et al. \(2009\)](#) for a review of the field of natural image statistics.

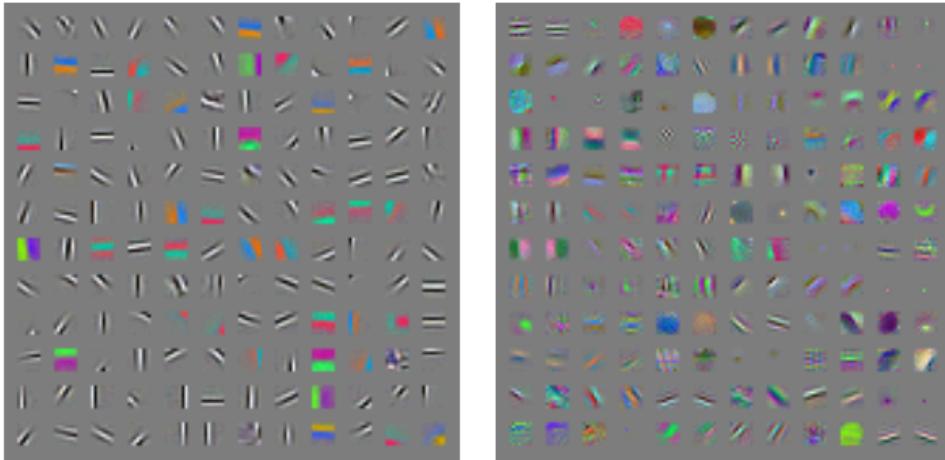


Figure 9.19: Many machine learning algorithms learn features that detect edges or specific colors of edges when applied to natural images. These feature detectors are reminiscent of the Gabor functions known to be present in the primary visual cortex. (*Left*)Weights learned by an unsupervised learning algorithm (spike and slab sparse coding) applied to small image patches. (*Right*)Convolution kernels learned by the first layer of a fully supervised convolutional maxout network. Neighboring pairs of filters drive the same maxout unit.

9.11 Convolutional Networks and the History of Deep Learning

Convolutional networks have played an important role in the history of deep learning. They are a key example of a successful application of insights obtained by studying the brain to machine learning applications. They were also some of the first deep models to perform well, long before arbitrary deep models were considered viable. Convolutional networks were also some of the first neural networks to solve important commercial applications and remain at the forefront of commercial applications of deep learning today. For example, in the 1990s, the neural network research group at AT&T developed a convolutional network for reading checks (LeCun *et al.*, 1998b). By the end of the 1990s, this system deployed by NCR was reading over 10 percent of all the checks in the United States. Later, several OCR and handwriting recognition systems based on convolutional nets were deployed by Microsoft (Simard *et al.*, 2003). See chapter 12 for more details on such applications and more modern applications of convolutional networks. See LeCun *et al.* (2010) for a more in-depth history of convolutional networks up to 2010.

Convolutional networks were also used to win many contests. The current intensity of commercial interest in deep learning began when Krizhevsky *et al.*

(2012) won the ImageNet object recognition challenge, but convolutional networks had been used to win other machine learning and computer vision contests with less impact for years earlier.

Convolutional nets were some of the first working deep networks trained with back-propagation. It is not entirely clear why convolutional networks succeeded when general back-propagation networks were considered to have failed. It may simply be that convolutional networks were more computationally efficient than fully connected networks, so it was easier to run multiple experiments with them and tune their implementation and hyperparameters. Larger networks also seem to be easier to train. With modern hardware, large fully connected networks appear to perform reasonably on many tasks, even when using datasets that were available and activation functions that were popular during the times when fully connected networks were believed not to work well. It may be that the primary barriers to the success of neural networks were psychological (practitioners did not expect neural networks to work, so they did not make a serious effort to use neural networks). Whatever the case, it is fortunate that convolutional networks performed well decades ago. In many ways, they carried the torch for the rest of deep learning and paved the way to the acceptance of neural networks in general.

Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size. This approach has been the most successful on a two-dimensional image topology. To process one-dimensional sequential data, we turn next to another powerful specialization of the neural networks framework: recurrent neural networks.

Chapter 10

Sequence Modeling: Recurrent and Recursive Nets

Recurrent neural networks, or RNNs ([Rumelhart *et al.*, 1986a](#)), are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values \mathbf{X} such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

To go from multilayer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters across different parts of a model. Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence. For example, consider the two sentences “I went to Nepal in 2009” and “In 2009, I went to Nepal.” If we ask a machine learning model to read each sentence and extract the year in which the narrator went to Nepal, we would like it to recognize the year 2009 as the relevant piece of information, whether it appears in the sixth

word or in the second word of the sentence. Suppose that we trained a feedforward network that processes sentences of fixed length. A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all the rules of the language separately at each position in the sentence. By comparison, a recurrent neural network shares the same weights across several time steps.

A related idea is the use of convolution across a 1-D temporal sequence. This convolutional approach is the basis for time-delay neural networks ([Lang and Hinton, 1988](#); [Waibel et al., 1989](#); [Lang et al., 1990](#)). The convolution operation allows a network to share parameters across time but is shallow. The output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input. The idea of parameter sharing manifests in the application of the same convolution kernel at each time step. Recurrent networks share parameters in a different way. Each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.

For the simplicity of exposition, we refer to RNNs as operating on a sequence that contains vectors $\mathbf{x}^{(t)}$ with the time step index t ranging from 1 to τ . In practice, recurrent networks usually operate on minibatches of such sequences, with a different sequence length τ for each member of the minibatch. We have omitted the minibatch indices to simplify notation. Moreover, the time step index need not literally refer to the passage of time in the real world. Sometimes it refers only to the position in the sequence. RNNs may also be applied in two dimensions across spatial data such as images, and even when applied to data involving time, the network may have connections that go backward in time, provided that the entire sequence is observed before it is provided to the network.

This chapter extends the idea of a computational graph to include cycles. These cycles represent the influence of the present value of a variable on its own value at a future time step. Such computational graphs allow us to define recurrent neural networks. We then describe many different ways to construct, train, and use recurrent neural networks.

For more information on recurrent neural networks than is available in this chapter, we refer the reader to the textbook of [Graves \(2012\)](#).

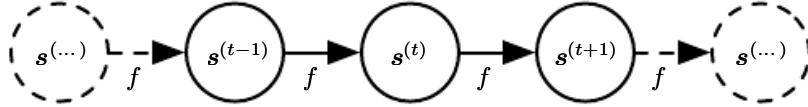


Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t , and the function f maps the state at t to the state at $t + 1$. The same parameters (the same value of θ used to parametrize f) are used for all time steps.

10.1 Unfolding Computational Graphs

A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Please refer to section 6.5.1 for a general introduction. In this section we explain the idea of **unfolding** a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure.

For example, consider the classical form of a dynamical system:

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \theta), \quad (10.1)$$

where $\mathbf{s}^{(t)}$ is called the state of the system.

Equation 10.1 is recurrent because the definition of \mathbf{s} at time t refers back to the same definition at time $t - 1$.

For a finite number of time steps τ , the graph can be unfolded by applying the definition $\tau - 1$ times. For example, if we unfold equation 10.1 for $\tau = 3$ time steps, we obtain

$$\mathbf{s}^{(3)} = f(\mathbf{s}^{(2)}; \theta) \quad (10.2)$$

$$= f(f(\mathbf{s}^{(1)}; \theta); \theta). \quad (10.3)$$

Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph. The unfolded computational graph of equation 10.1 and equation 10.3 is illustrated in figure 10.1.

As another example, let us consider a dynamical system driven by an external

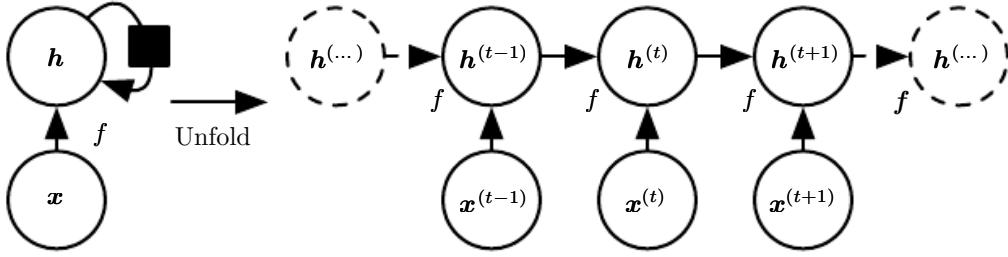


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input \mathbf{x} by incorporating it into the state \mathbf{h} that is passed forward through time. (*Left*) Circuit diagram. The black square indicates a delay of a single time step. (*Right*) The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

signal $\mathbf{x}^{(t)}$,

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}), \quad (10.4)$$

where we see that the state now contains information about the whole past sequence.

Recurrent neural networks can be built in many different ways. Much as almost any function can be considered a feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network.

Many recurrent neural networks use equation 10.5 or a similar equation to define the values of their hidden units. To indicate that the state is the hidden units of the network, we now rewrite equation 10.4 using the variable \mathbf{h} to represent the state,

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}), \quad (10.5)$$

illustrated in figure 10.2; typical RNNs will add extra architectural features such as output layers that read information out of the state \mathbf{h} to make predictions.

When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\mathbf{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t . This summary is in general necessarily lossy, since it maps an arbitrary length sequence $(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$ to a fixed length vector $\mathbf{h}^{(t)}$. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects. For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, storing all the information in the input sequence up to time t may not be necessary; storing only enough information to predict the rest of the sentence is sufficient. The most demanding situation is when we ask $\mathbf{h}^{(t)}$ to be rich enough to allow one to approximately recover the input sequence, as in autoencoder

frameworks (chapter 14).

Equation 10.5 can be drawn in two different ways. One way to draw the RNN is with a diagram containing one node for every component that might exist in a physical implementation of the model, such as a biological neural network. In this view, the network defines a circuit that operates in real time, with physical parts whose current state can influence their future state, as in the left of figure 10.2. Throughout this chapter, we use a black square in a circuit diagram to indicate that an interaction takes place with a delay of a single time step, from the state at time t to the state at time $t + 1$. The other way to draw the RNN is as an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that point in time. Each variable for each time step is drawn as a separate node of the computational graph, as in the right of figure 10.2. What we call unfolding is the operation that maps a circuit, as in the left side of the figure, to a computational graph with repeated pieces, as in the right side. The unfolded graph now has a size that depends on the sequence length.

We can represent the unfolded recurrence after t steps with a function $g^{(t)}$:

$$\mathbf{h}^{(t)} = g^{(t)}(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}) \quad (10.6)$$

$$= f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}). \quad (10.7)$$

The function $g^{(t)}$ takes the whole past sequence $(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f . The unfolding process thus introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
2. It is possible to use the *same* transition function f with the same parameters at every time step.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model $g^{(t)}$ for all possible time steps. Learning a single shared model allows generalization to sequence lengths that did not appear in the training set, and enables the model to be estimated with far fewer training examples than would be required without parameter sharing.

Both the recurrent graph and the unrolled graph have their uses. The recurrent graph is succinct. The unfolded graph provides an explicit description of which computations to perform. The unfolded graph also helps illustrate the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.

10.2 Recurrent Neural Networks

Armed with the graph-unrolling and parameter-sharing ideas of section 10.1, we can design a wide variety of recurrent neural networks.

Some examples of important design patterns for recurrent neural networks include the following:

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in figure 10.3
- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step, illustrated in figure 10.4
- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output, illustrated in figure 10.5

Figure 10.3 is a reasonably representative example that we return to throughout most of the chapter.

The recurrent neural network of figure 10.3 and equation 10.8 is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size. The output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input (Siegelmann and Sontag, 1991; Siegelmann, 1995; Siegelmann and Sontag, 1995; Hyötyniemi, 1996). The functions computable by a Turing machine are discrete, so these results regard exact implementation of the function, not approximations. The RNN, when used as a Turing machine, takes a binary sequence as input, and its outputs must be discretized to provide a binary output. It is possible to compute all functions in this setting using a single specific RNN of finite size (Siegelmann and Sontag[1995] use 886 units). The “input” of the Turing machine is

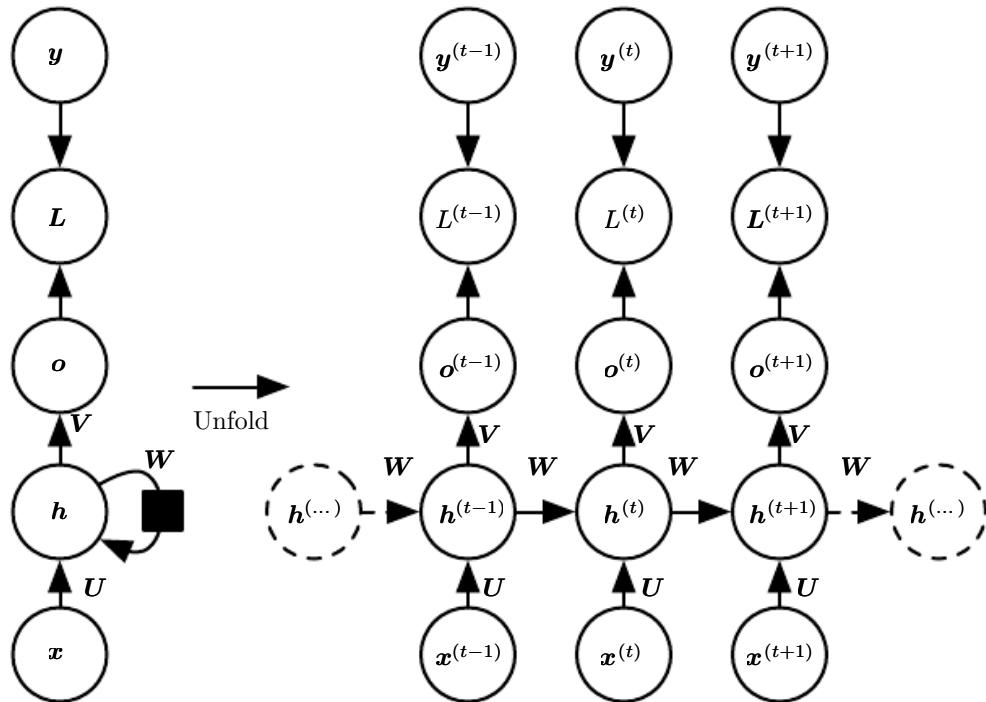


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of \mathbf{x} values to a corresponding sequence of output \mathbf{o} values. A loss L measures how far each \mathbf{o} is from the corresponding training target \mathbf{y} . When using softmax outputs, we assume \mathbf{o} is the unnormalized log probabilities. The loss L internally computes $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ and compares this to the target \mathbf{y} . The RNN has input to hidden connections parametrized by a weight matrix \mathbf{U} , hidden-to-hidden recurrent connections parametrized by a weight matrix \mathbf{W} , and hidden-to-output connections parametrized by a weight matrix \mathbf{V} . Equation 10.8 defines forward propagation in this model. (*Left*) The RNN and its loss drawn with recurrent connections. (*Right*) The same seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.

a specification of the function to be computed, so the same network that simulates this Turing machine is sufficient for all problems. The theoretical RNN used for the proof can simulate an unbounded stack by representing its activations and weights with rational numbers of unbounded precision.

We now develop the forward propagation equations for the RNN depicted in figure 10.3. The figure does not specify the choice of activation function for the hidden units. Here we assume the hyperbolic tangent activation function. Also, the figure does not specify exactly what form the output and loss function take. Here we assume that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent discrete variables is to regard the output \mathbf{o} as giving the unnormalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector $\hat{\mathbf{y}}$ of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $\mathbf{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}, \quad (10.8)$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}), \quad (10.9)$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}, \quad (10.10)$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}), \quad (10.11)$$

where the parameters are the bias vectors \mathbf{b} and \mathbf{c} along with the weight matrices \mathbf{U} , \mathbf{V} and \mathbf{W} , respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of \mathbf{x} values paired with a sequence of \mathbf{y} values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}$, then

$$L\left(\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}\}\right) \quad (10.12)$$

$$= \sum_t L^{(t)} \quad (10.13)$$

$$= - \sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right), \quad (10.14)$$

where $p_{\text{model}}\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right)$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\mathbf{y}}^{(t)}$. Computing the gradient of this loss function with respect to the parameters is an expensive operation. The gradient computation involves performing a forward propagation pass moving left to right through our illustration

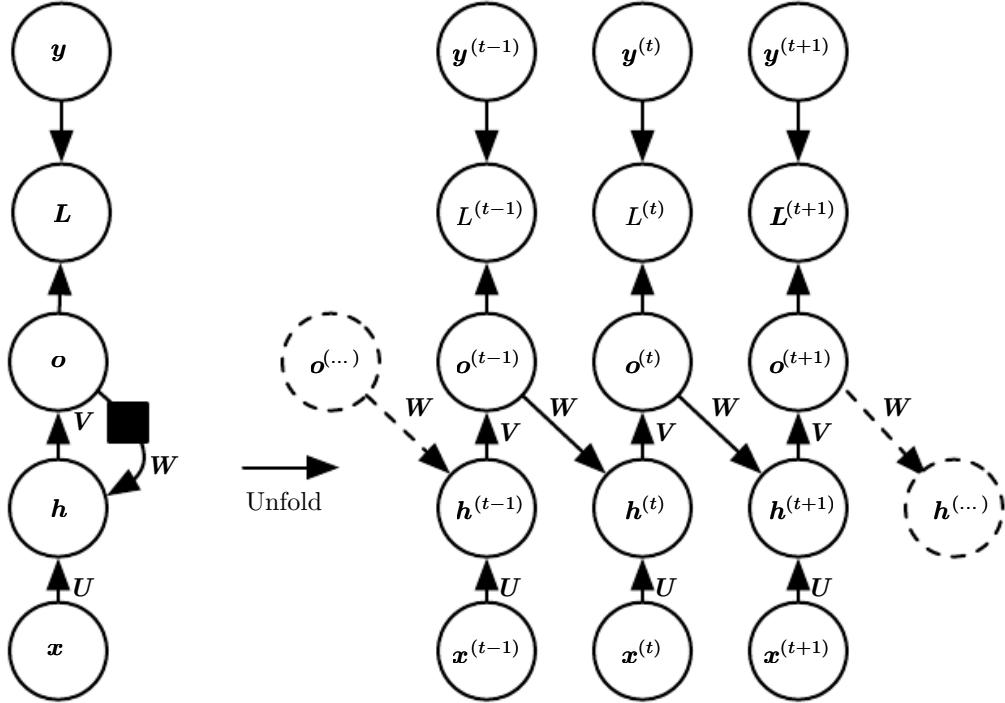


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t , the input is \mathbf{x}_t , the hidden layer activations are $\mathbf{h}^{(t)}$, the outputs are $\mathbf{o}^{(t)}$, the targets are $\mathbf{y}^{(t)}$, and the loss is $L^{(t)}$. (Left) Circuit diagram. (Right) Unfolded computational graph. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by figure 10.3. The RNN in figure 10.3 can choose to put any information it wants about the past into its hidden representation \mathbf{h} and transmit \mathbf{h} to the future. The RNN in this figure is trained to put a specific output value into \mathbf{o} , and \mathbf{o} is the only information it is allowed to send to the future. There are no direct connections from \mathbf{h} going forward. The previous \mathbf{h} is connected to the present only indirectly, via the predictions it was used to produce. Unless \mathbf{o} is very high-dimensional and rich, it will usually lack important information from the past. This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training, as described in section 10.2.1.

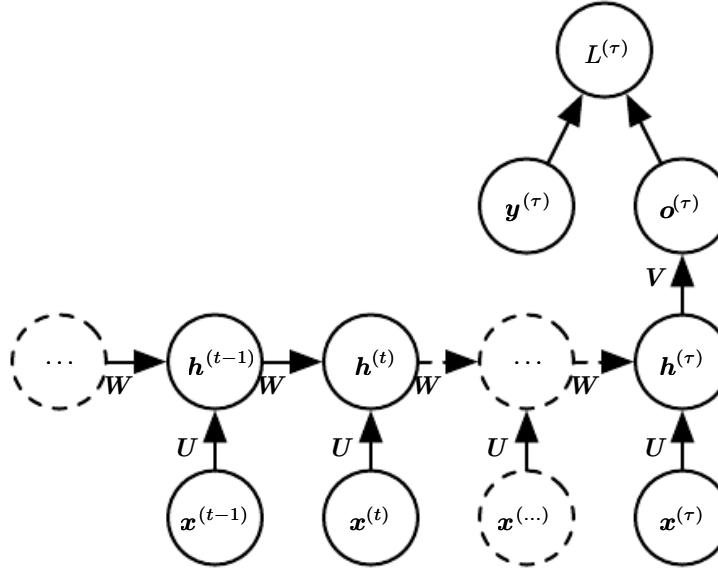


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here), or the gradient on the output $o^{(\tau)}$ can be obtained by back-propagating from further downstream modules.

of the unrolled graph in figure 10.3, followed by a backward propagation pass moving right to left through the graph. The runtime is $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$. The back-propagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called **back-propagation through time** (BPTT) and is discussed further in section 10.2.2. The network with recurrence between hidden units is thus very powerful but also expensive to train. Is there an alternative?

10.2.1 Teacher Forcing and Networks with Output Recurrence

The network with recurrent connections only from the output at one time step to the hidden units at the next time step (shown in figure 10.4) is strictly less powerful because it lacks hidden-to-hidden recurrent connections. For example, it cannot simulate a universal Turing machine. Because this network lacks hidden-to-hidden recurrence, it requires that the output units capture all the information about the past that the network will use to predict the future. Because the output units are

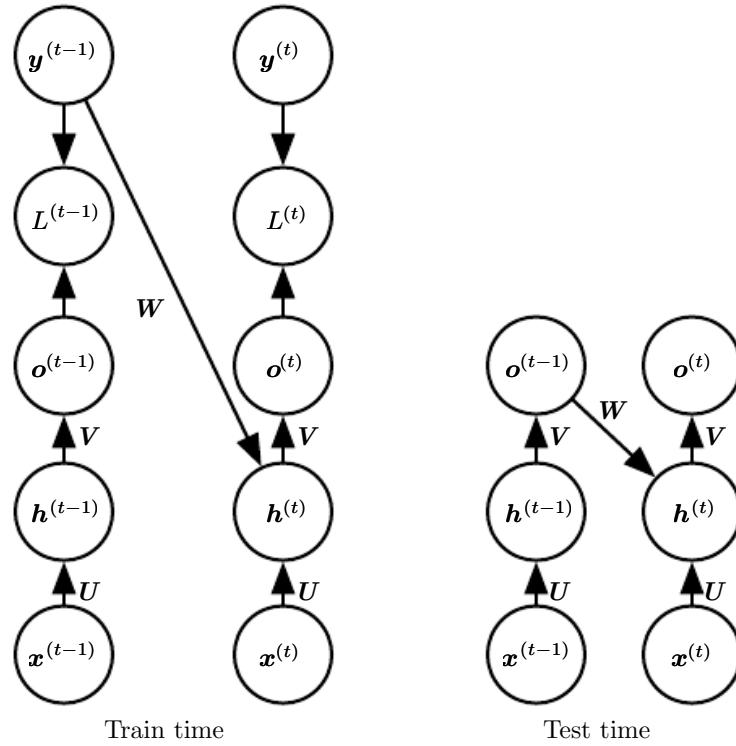


Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. (*Left*) At train time, we feed the *correct output* $y^{(t)}$ drawn from the train set as input to $h^{(t+1)}$. (*Right*) When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $y^{(t)}$ with the model’s output $o^{(t)}$, and feed the output back into the model.

explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets. The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t , all the time steps are decoupled. Training can thus be parallelized, with the gradient for each step t computed in isolation. There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

Models that have recurrent connections from their outputs leading back into the model may be trained with **teacher forcing**. Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$. We can see this by examining a sequence with two time steps. The conditional maximum

likelihood criterion is

$$\log p(\mathbf{y}^{(1)}, \mathbf{y}^{(2)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) \quad (10.15)$$

$$= \log p(\mathbf{y}^{(2)} | \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) + \log p(\mathbf{y}^{(1)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}). \quad (10.16)$$

In this example, we see that at time $t = 2$, the model is trained to maximize the conditional probability of $\mathbf{y}^{(2)}$ given *both* the \mathbf{x} sequence so far and the previous \mathbf{y} value from the training set. Maximum likelihood thus specifies that during training, rather than feeding the model’s own output back into itself, these connections should be fed with the target values specifying what the correct output should be. This is illustrated in figure 10.6.

We originally motivated teacher forcing as allowing us to avoid back-propagation through time in models that lack hidden-to-hidden connections. Teacher forcing may still be applied to models that have hidden-to-hidden connections as long as they have connections from the output at one time step to values computed in the next time step. As soon as the hidden units become a function of earlier time steps, however, the BPTT algorithm is necessary. Some models may thus be trained with both teacher forcing and BPTT.

The disadvantage of strict teacher forcing arises if the network is going to be later used in an **closed-loop** mode, with the network outputs (or samples from the output distribution) fed back as input. In this case, the fed-back inputs that the network sees during training could be quite different from the kind of inputs that it will see at test time. One way to mitigate this problem is to train with both teacher-forced inputs and free-running inputs, for example by predicting the correct target a number of steps in the future through the unfolded recurrent output-to-input paths. In this way, the network can learn to take into account input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back toward one that will make the network generate proper outputs after a few steps. Another approach (Bengio *et al.*, 2015b) to mitigate the gap between the inputs seen at training time and the inputs seen at test time randomly chooses to use generated values or actual data values as input. This approach exploits a curriculum learning strategy to gradually use more of the generated values as input.

10.2.2 Computing the Gradient in a Recurrent Neural Network

Computing the gradient through a recurrent neural network is straightforward. One simply applies the generalized back-propagation algorithm of section 6.5.6 to the unrolled computational graph. No specialized algorithms are necessary.

Gradients obtained by back-propagation may then be used with any general-purpose gradient-based techniques to train an RNN.

To gain some intuition for how the BPTT algorithm behaves, we provide an example of how to compute gradients by BPTT for the RNN equations above (equation 10.8 and equation 10.12). The nodes of our computational graph include the parameters \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} and \mathbf{c} as well as the sequence of nodes indexed by t for $\mathbf{x}^{(t)}$, $\mathbf{h}^{(t)}$, $\mathbf{o}^{(t)}$ and $L^{(t)}$. For each node \mathbf{N} we need to compute the gradient $\nabla_{\mathbf{N}} L$ recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss:

$$\frac{\partial L}{\partial L^{(t)}} = 1. \quad (10.17)$$

In this derivation we assume that the outputs $\mathbf{o}^{(t)}$ are used as the argument to the softmax function to obtain the vector $\hat{\mathbf{y}}$ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target $y^{(t)}$ given the input so far. The gradient $\nabla_{\mathbf{o}^{(t)}} L$ on the outputs at time step t , for all i, t , is as follows:

$$(\nabla_{\mathbf{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i=y^{(t)}}. \quad (10.18)$$

We work our way backward, starting from the end of the sequence. At the final time step τ , $\mathbf{h}^{(\tau)}$ only has $\mathbf{o}^{(\tau)}$ as a descendent, so its gradient is simple:

$$\nabla_{\mathbf{h}^{(\tau)}} L = \mathbf{V}^\top \nabla_{\mathbf{o}^{(\tau)}} L. \quad (10.19)$$

We can then iterate backward in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $\mathbf{h}^{(t)}$ (for $t < \tau$) has as descendants both $\mathbf{o}^{(t)}$ and $\mathbf{h}^{(t+1)}$. Its gradient is thus given by

$$\nabla_{\mathbf{h}^{(t)}} L = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{h}^{(t+1)}} L) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{o}^{(t)}} L) \quad (10.20)$$

$$= \mathbf{W}^\top \text{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right) (\nabla_{\mathbf{h}^{(t+1)}} L) + \mathbf{V}^\top (\nabla_{\mathbf{o}^{(t)}} L), \quad (10.21)$$

where $\text{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right)$ indicates the diagonal matrix containing the elements $1 - (h_i^{(t+1)})^2$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t + 1$.

Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes. Because the

parameters are shared across many time steps, we must take some care when denoting calculus operations involving these variables. The equations we wish to implement use the `bprop` method of section 6.5.6, which computes the contribution of a single edge in the computational graph to the gradient. The $\nabla_{\mathbf{W}} f$ operator used in calculus, however, takes into account the contribution of \mathbf{W} to the value of f due to *all* edges in the computational graph. To resolve this ambiguity, we introduce dummy variables $\mathbf{W}^{(t)}$ that are defined to be copies of \mathbf{W} but with each $\mathbf{W}^{(t)}$ used only at time step t . We may then use $\nabla_{\mathbf{W}^{(t)}}$ to denote the contribution of the weights at time step t to the gradient.

Using this notation, the gradient on the remaining parameters is given by

$$\nabla_{\mathbf{c}} L = \sum_t \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}} \right)^{\top} \nabla_{\mathbf{o}^{(t)}} L = \sum_t \nabla_{\mathbf{o}^{(t)}} L, \quad (10.22)$$

$$\nabla_{\mathbf{b}} L = \sum_t \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}^{(t)}} \right)^{\top} \nabla_{\mathbf{h}^{(t)}} L = \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) \nabla_{\mathbf{h}^{(t)}} L, \quad (10.23)$$

$$\nabla_{\mathbf{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\mathbf{V}^{(t)}} o_i^{(t)} = \sum_t (\nabla_{\mathbf{o}^{(t)}} L) \mathbf{h}^{(t)\top}, \quad (10.24)$$

$$\begin{aligned} \nabla_{\mathbf{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{W}^{(t)}} h_i^{(t)} \\ &= \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{h}^{(t-1)\top}, \end{aligned} \quad (10.25)$$

$$\nabla_{\mathbf{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{U}^{(t)}} h_i^{(t)} \quad (10.27)$$

$$= \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{x}^{(t)\top}, \quad (10.28)$$

We do not need to compute the gradient with respect to $\mathbf{x}^{(t)}$ for training because it does not have any parameters as ancestors in the computational graph defining the loss.

10.2.3 Recurrent Networks as Directed Graphical Models

In the example recurrent network we have developed so far, the losses $L^{(t)}$ were cross-entropies between training targets $\mathbf{y}^{(t)}$ and outputs $\mathbf{o}^{(t)}$. As with a feedforward network, it is in principle possible to use almost any loss with a recurrent network.

The loss should be chosen based on the task. As with a feedforward network, we usually wish to interpret the output of the RNN as a probability distribution, and we usually use the cross-entropy associated with that distribution to define the loss. Mean squared error is the cross-entropy loss associated with an output distribution that is a unit Gaussian, for example, just as with a feedforward network.

When we use a predictive log-likelihood training objective, such as equation 10.12, we train the RNN to estimate the conditional distribution of the next sequence element $\mathbf{y}^{(t)}$ given the past inputs. This may mean that we maximize the log-likelihood

$$\log p(\mathbf{y}^{(t)} \mid \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}), \quad (10.29)$$

or, if the model includes connections from the output at one time step to the next time step,

$$\log p(\mathbf{y}^{(t)} \mid \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(t-1)}). \quad (10.30)$$

Decomposing the joint probability over the sequence of \mathbf{y} values as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence. When we do not feed past \mathbf{y} values as inputs that condition the next step prediction, the directed graphical model contains no edges from any $\mathbf{y}^{(i)}$ in the past to the current $\mathbf{y}^{(t)}$. In this case, the outputs \mathbf{y} are conditionally independent given the sequence of \mathbf{x} values. When we do feed the actual \mathbf{y} values (not their prediction, but the actual observed or generated values) back into the network, the directed graphical model contains edges from all $\mathbf{y}^{(i)}$ values in the past to the current $\mathbf{y}^{(t)}$ value.

As a simple example, let us consider the case where the RNN models only a sequence of scalar random variables $\mathbb{Y} = \{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}\}$, with no additional inputs \mathbf{x} . The input at time step t is simply the output at time step $t - 1$. The RNN then defines a directed graphical model over the \mathbf{y} variables. We parametrize the joint distribution of these observations using the chain rule (equation 3.6) for conditional probabilities:

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \dots, \mathbf{y}^{(1)}), \quad (10.31)$$

where the righthand side of the bar is empty for $t = 1$, of course. Hence the negative log-likelihood of a set of values $\{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}\}$ according to such a model is

$$L = \sum_t L^{(t)}, \quad (10.32)$$

where

$$L^{(t)} = -\log P(\mathbf{y}^{(t)} = \mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \dots, \mathbf{y}^{(1)}). \quad (10.33)$$

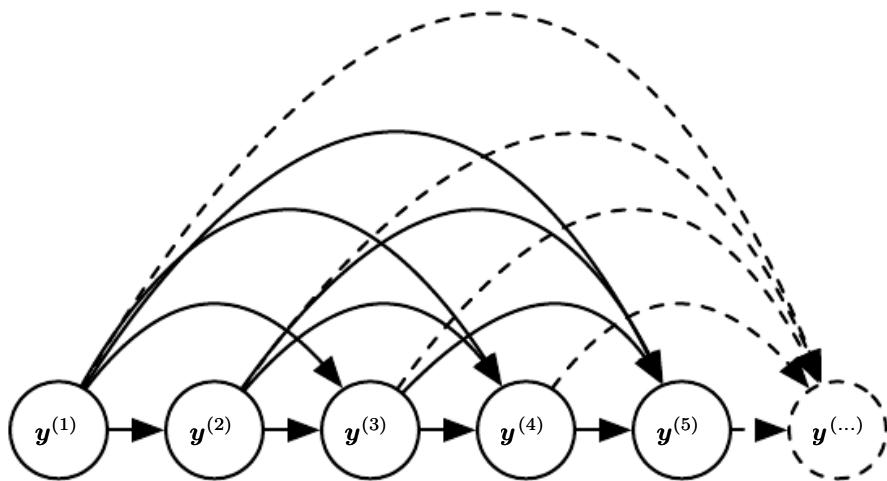


Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$. Every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (as in equation 10.6) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in figure 10.8.

The edges in a graphical model indicate which variables depend directly on other variables. Many graphical models aim to achieve statistical and computational efficiency by omitting edges that do not correspond to strong interactions. For example, it is common to make the Markov assumption that the graphical model should contain only edges from $\{y^{(t-k)}, \dots, y^{(t-1)}\}$ to $y^{(t)}$, rather than containing edges from the entire history. In some cases, however, we believe that all past inputs should have an influence on the next element of the sequence. RNNs are useful when we believe that the distribution over $y^{(t)}$ may depend on a value of $y^{(i)}$ from the distant past in a way that is not captured by the effect of $y^{(i)}$ on $y^{(t-1)}$.

One way to interpret an RNN as a graphical model is to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of y values. The graphical model over the y values with the complete graph structure is shown in figure 10.7. The complete graph interpretation of the RNN is based on ignoring the hidden units $\mathbf{h}^{(t)}$ by marginalizing them out of the model.

It is more interesting to consider the graphical model structure of RNNs that results from regarding the hidden units $\mathbf{h}^{(t)}$ as random variables.¹ Including the

¹The conditional distribution over these variables given their parents is deterministic. This is perfectly legitimate, though it is somewhat rare to design a graphical model with such deterministic hidden units.

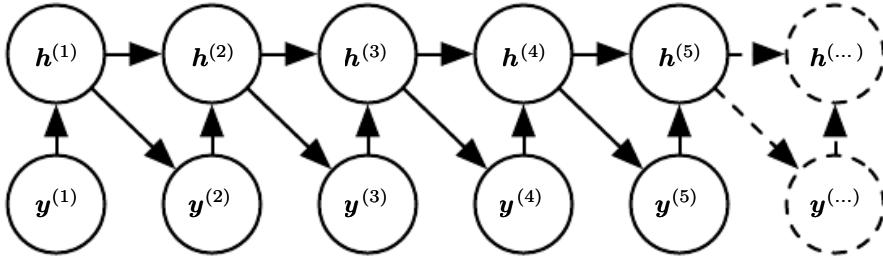


Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $\mathbf{h}^{(t)}$ and $\mathbf{y}^{(t)}$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

hidden units in the graphical model reveals that the RNN provides an efficient parametrization of the joint distribution over the observations. Suppose that we represented an arbitrary joint distribution over discrete values with a tabular representation—an array containing a separate entry for each possible assignment of values, with the value of that entry giving the probability of that assignment occurring. If y can take on k different values, the tabular representation would have $O(k^T)$ parameters. By comparison, because of parameter sharing, the number of parameters in the RNN is $O(1)$ as a function of sequence length. The number of parameters in the RNN may be adjusted to control model capacity but is not forced to scale with sequence length. Equation 10.5 shows that the RNN parametrizes long-term relationships between variables efficiently, using recurrent applications of the same function f and the same parameters θ at each time step. Figure 10.8 illustrates the graphical model interpretation. Incorporating the $\mathbf{h}^{(t)}$ nodes in the graphical model decouples the past and the future, acting as an intermediate quantity between them. A variable $y^{(i)}$ in the distant past may influence a variable $y^{(t)}$ via its effect on \mathbf{h} . The structure of this graph shows that the model can be efficiently parametrized by using the same conditional probability distributions at each time step, and that when the variables are all observed, the probability of the joint assignment of all variables can be evaluated efficiently.

Even with the efficient parametrization of the graphical model, some operations remain computationally challenging. For example, it is difficult to predict missing values in the middle of the sequence.

The price recurrent networks pay for their reduced number of parameters is that *optimizing* the parameters may be difficult.

The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps. Equivalently, the

assumption is that the conditional probability distribution over the variables at time $t+1$ given the variables at time t is **stationary**, meaning that the relationship between the previous time step and the next time step does not depend on t . In principle, it would be possible to use t as an extra input at each time step and let the learner discover any time-dependence while sharing as much as it can between different time steps. This would already be much better than using a different conditional probability distribution for each t , but the network would then have to extrapolate when faced with new values of t .

To complete our view of an RNN as a graphical model, we must describe how to draw samples from the model. The main operation that we need to perform is simply to sample from the conditional distribution at each time step. However, there is one additional complication. The RNN must have some mechanism for determining the length of the sequence. This can be achieved in various ways.

When the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence (Schmidhuber, 2012). When that symbol is generated, the sampling process stops. In the training set, we insert this symbol as an extra member of the sequence, immediately after $\mathbf{x}^{(\tau)}$ in each training example.

Another option is to introduce an extra Bernoulli output to the model that represents the decision to either continue generation or halt generation at each time step. This approach is more general than the approach of adding an extra symbol to the vocabulary, because it may be applied to any RNN, rather than only RNNs that output a sequence of symbols. For example, it may be applied to an RNN that emits a sequence of real numbers. The new output unit is usually a sigmoid unit trained with the cross-entropy loss. In this approach the sigmoid is trained to maximize the log-probability of the correct prediction as to whether the sequence ends or continues at each time step.

Another way to determine the sequence length τ is to add an extra output to the model that predicts the integer τ itself. The model can sample a value of τ and then sample τ steps worth of data. This approach requires adding an extra input to the recurrent update at each time step so that the recurrent update is aware of whether it is near the end of the generated sequence. This extra input can either consist of the value of τ or can consist of $\tau - t$, the number of remaining time steps. Without this extra input, the RNN might generate sequences that end abruptly, such as a sentence that ends before it is complete. This approach is based on the decomposition

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}) = P(\tau)P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)} | \tau). \quad (10.34)$$

The strategy of predicting τ directly is used, for example, by Goodfellow *et al.* (2014d).

10.2.4 Modeling Sequences Conditioned on Context with RNNs

In the previous section we described how an RNN could correspond to a directed graphical model over a sequence of random variables $y^{(t)}$ with no inputs \mathbf{x} . Of course, our development of RNNs as in equation 10.8 included a sequence of inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(\tau)}$. In general, RNNs allow the extension of the graphical model view to represent not only a joint distribution over the y variables but also a conditional distribution over y given \mathbf{x} . As discussed in the context of feedforward networks in section 6.2.1.1, any model representing a variable $P(\mathbf{y}; \boldsymbol{\theta})$ can be reinterpreted as a model representing a conditional distribution $P(\mathbf{y}|\boldsymbol{\omega})$ with $\boldsymbol{\omega} = \boldsymbol{\theta}$. We can extend such a model to represent a distribution $P(\mathbf{y}|\mathbf{x})$ by using the same $P(\mathbf{y}|\boldsymbol{\omega})$ as before, but making $\boldsymbol{\omega}$ a function of \mathbf{x} . In the case of an RNN, this can be achieved in different ways. We review here the most common and obvious choices.

Previously, we have discussed RNNs that take a sequence of vectors $\mathbf{x}^{(t)}$ for $t = 1, \dots, \tau$ as input. Another option is to take only a single vector \mathbf{x} as input. When \mathbf{x} is a fixed-size vector, we can simply make it an extra input of the RNN that generates the \mathbf{y} sequence. Some common ways of providing an extra input to an RNN are

1. as an extra input at each time step, or
2. as the initial state $\mathbf{h}^{(0)}$, or
3. both.

The first and most common approach is illustrated in figure 10.9. The interaction between the input \mathbf{x} and each hidden unit vector $\mathbf{h}^{(t)}$ is parametrized by a newly introduced weight matrix \mathbf{R} that was absent from the model of only the sequence of y values. The same product $\mathbf{x}^\top \mathbf{R}$ is added as additional input to the hidden units at every time step. We can think of the choice of \mathbf{x} as determining the value of $\mathbf{x}^\top \mathbf{R}$ that is effectively a new bias parameter used for each of the hidden units. The weights remain independent of the input. We can think of this model as taking the parameters $\boldsymbol{\theta}$ of the nonconditional model and turning them into $\boldsymbol{\omega}$, where the bias parameters within $\boldsymbol{\omega}$ are now a function of the input.

Rather than receiving only a single vector \mathbf{x} as input, the RNN may receive a sequence of vectors $\mathbf{x}^{(t)}$ as input. The RNN described in equation 10.8 corre-

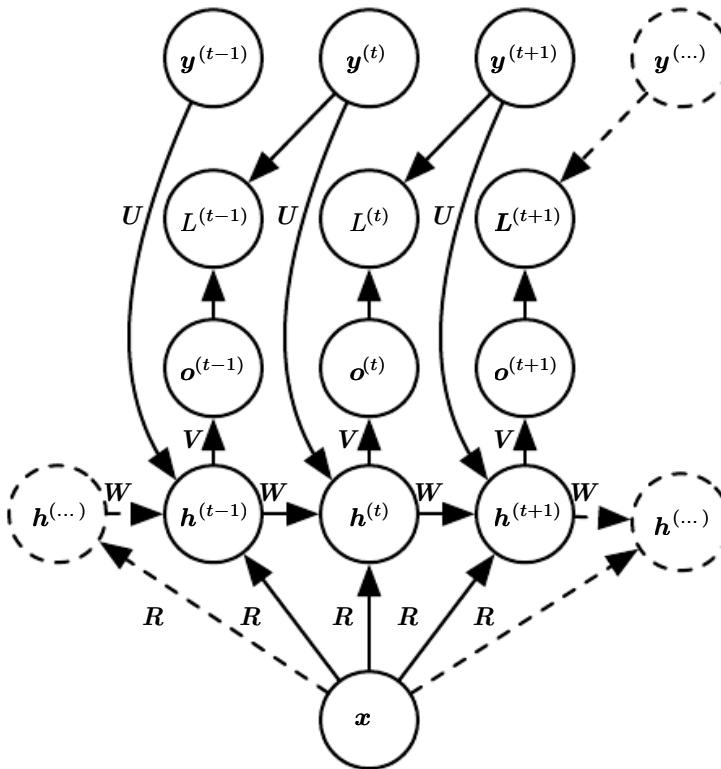


Figure 10.9: An RNN that maps a fixed-length vector \mathbf{x} into a distribution over sequences \mathbf{Y} . This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element $y^{(t)}$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

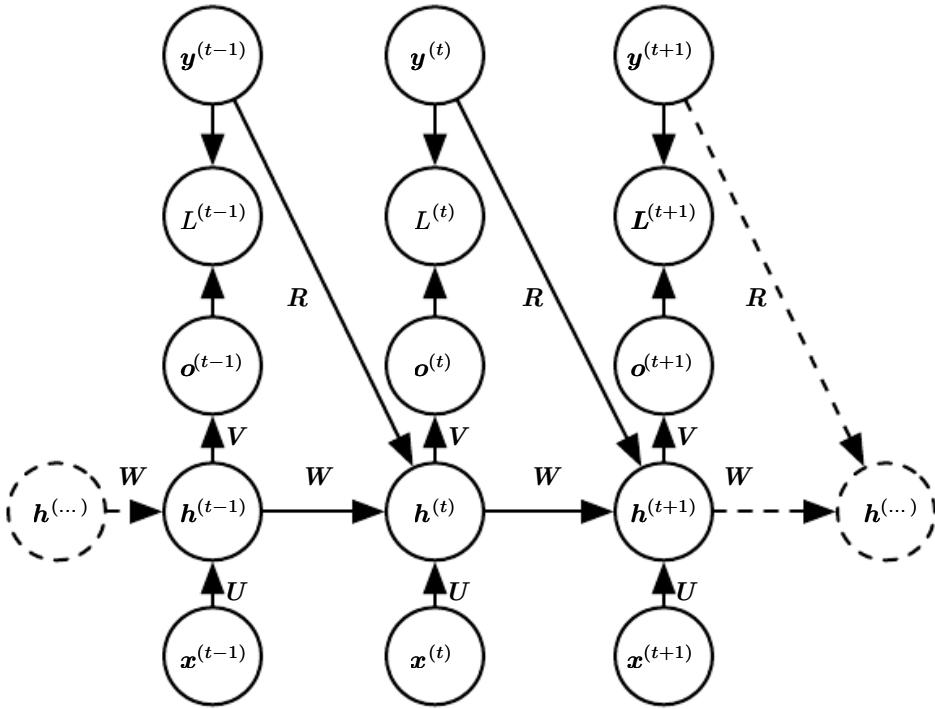


Figure 10.10: A conditional recurrent neural network mapping a variable-length sequence of \mathbf{x} values into a distribution over sequences of \mathbf{y} values of the same length. Compared to figure 10.3, this RNN contains connections from the previous output to the current state. These connections allow this RNN to model an arbitrary distribution over sequences of \mathbf{y} given sequences of \mathbf{x} of the same length. The RNN of figure 10.3 is only able to represent distributions in which the \mathbf{y} values are conditionally independent from each other given the \mathbf{x} values.

sponds to a conditional distribution $P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)})$ that makes a conditional independence assumption that this distribution factorizes as

$$\prod_t P(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}). \quad (10.35)$$

To remove the conditional independence assumption, we can add connections from the output at time t to the hidden unit at time $t + 1$, as shown in figure 10.10. The model can then represent arbitrary probability distributions over the \mathbf{y} sequence. This kind of model representing a distribution over a sequence given another sequence still has one restriction, which is that the length of both sequences must be the same. We describe how to remove this restriction in section 10.4.

10.3 Bidirectional RNNs

All the recurrent networks we have considered up to now have a “causal” structure, meaning that the state at time t captures only information from the past, $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t-1)}$, and the present input $\mathbf{x}^{(t)}$. Some of the models we have discussed also allow information from past \mathbf{y} values to affect the current state when the \mathbf{y} values are available.

In many applications, however, we want to output a prediction of $\mathbf{y}^{(t)}$ that may depend on *the whole input sequence*. For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them. This is also true of handwriting recognition and many other sequence-to-sequence learning tasks, described in the next section.

Bidirectional recurrent neural networks (or bidirectional RNNs) were invented to address that need (Schuster and Paliwal, 1997). They have been extremely successful (Graves, 2012) in applications where that need arises, such as handwriting recognition (Graves *et al.*, 2008; Graves and Schmidhuber, 2009), speech recognition (Graves and Schmidhuber, 2005; Graves *et al.*, 2013), and bioinformatics (Baldi *et al.*, 1999).

As the name suggests, bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence. Figure 10.11 illustrates the typical bidirectional RNN, with $\mathbf{h}^{(t)}$ standing for the state of the

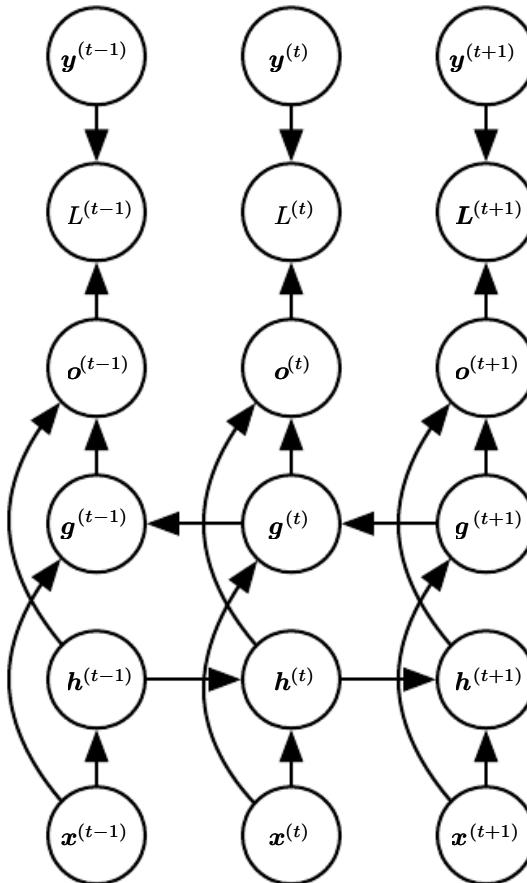


Figure 10.11: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences \mathbf{x} to target sequences \mathbf{y} , with loss $L^{(t)}$ at each step t . The \mathbf{h} recurrence propagates information forward in time (toward the right), while the \mathbf{g} recurrence propagates information backward in time (toward the left). Thus at each point t , the output units $\mathbf{o}^{(t)}$ can benefit from a relevant summary of the past in its $\mathbf{h}^{(t)}$ input and from a relevant summary of the future in its $\mathbf{g}^{(t)}$ input.

sub-RNN that moves forward through time and $\mathbf{g}^{(t)}$ standing for the state of the sub-RNN that moves backward through time. This allows the output units $\mathbf{o}^{(t)}$ to compute a representation that depends on *both the past and the future* but is most sensitive to the input values around time t , without having to specify a fixed-size window around t (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

This idea can be naturally extended to two-dimensional input, such as images, by having *four* RNNs, each one going in one of the four directions: up, down, left, right. At each point (i, j) of a 2-D grid, an output $O_{i,j}$ could then compute a representation that would capture mostly local information but could also depend on long-range inputs, if the RNN is able to learn to carry that information. Compared to a convolutional network, RNNs applied to images are typically more expensive but allow for long-range lateral interactions between features in the same feature map (Visin *et al.*, 2015; Kalchbrenner *et al.*, 2015). Indeed, the forward propagation equations for such RNNs may be written in a form that shows they use a convolution that computes the bottom-up input to each layer, prior to the recurrent propagation across the feature map that incorporates the lateral interactions.

10.4 Encoder-Decoder Sequence-to-Sequence Architectures

We have seen in figure 10.5 how an RNN can map an input sequence to a fixed-size vector. We have seen in figure 10.9 how an RNN can map a fixed-size vector to a sequence. We have seen in figures 10.3, 10.4, 10.10 and 10.11 how an RNN can map an input sequence to an output sequence of the same length.

Here we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length. This comes up in many applications, such as speech recognition, machine translation and question answering, where the input and output sequences in the training set are generally not of the same length (although their lengths might be related).

We often call the input to the RNN the “context.” We want to produce a representation of this context, C . The context C might be a vector or sequence of vectors that summarize the input sequence $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$.

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed by Cho *et al.* (2014a) and shortly after by Sutskever *et al.* (2014), who independently developed that archi-

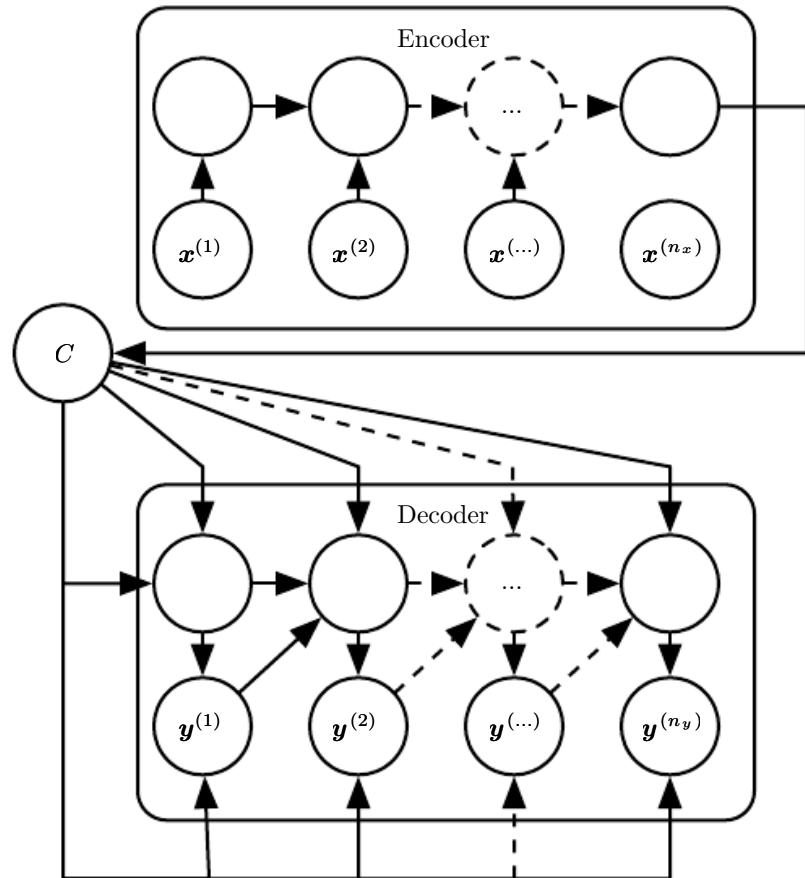


Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)})$ given an input sequence $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n_x)})$. It is composed of an encoder RNN that reads the input sequence as well as a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C , which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

ture and were the first to obtain state-of-the-art translation using this approach. The former system is based on scoring proposals generated by another machine translation system, while the latter uses a standalone recurrent network to generate the translations. These authors respectively called this architecture, illustrated in figure 10.12, the encoder-decoder or sequence-to-sequence architecture. The idea is very simple: (1) An **encoder** or **reader** or **input** RNN processes the input sequence. The encoder emits the context C , usually as a simple function of its final hidden state. (2) A **decoder** or **writer** or **output** RNN is conditioned on that fixed-length vector (just as in figure 10.9) to generate the output sequence $\mathbf{Y} = (\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)})$. The innovation of this kind of architecture over those presented in earlier sections of this chapter is that the lengths n_x and n_y can vary from each other, while previous architectures constrained $n_x = n_y = \tau$. In a sequence-to-sequence architecture, the two RNNs are trained jointly to maximize the average of $\log P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$ over all the pairs of \mathbf{x} and \mathbf{y} sequences in the training set. The last state \mathbf{h}_{n_x} of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.

If the context C is a vector, then the decoder RNN is simply a vector-to-sequence RNN, as described in section 10.2.4. As we have seen, there are at least two ways for a vector-to-sequence RNN to receive input. The input can be provided as the initial state of the RNN, or the input can be connected to the hidden units at each time step. These two ways can also be combined.

There is no constraint that the encoder must have the same size of hidden layer as the decoder.

One clear limitation of this architecture is when the context C output by the encoder RNN has a dimension that is too small to properly summarize a long sequence. This phenomenon was observed by Bahdanau *et al.* (2015) in the context of machine translation. They proposed to make C a variable-length sequence rather than a fixed-size vector. Additionally, they introduced an **attention mechanism** that learns to associate elements of the sequence C to elements of the output sequence. See section 12.4.5.1 for more details.

10.5 Deep Recurrent Networks

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. from the input to the hidden state,

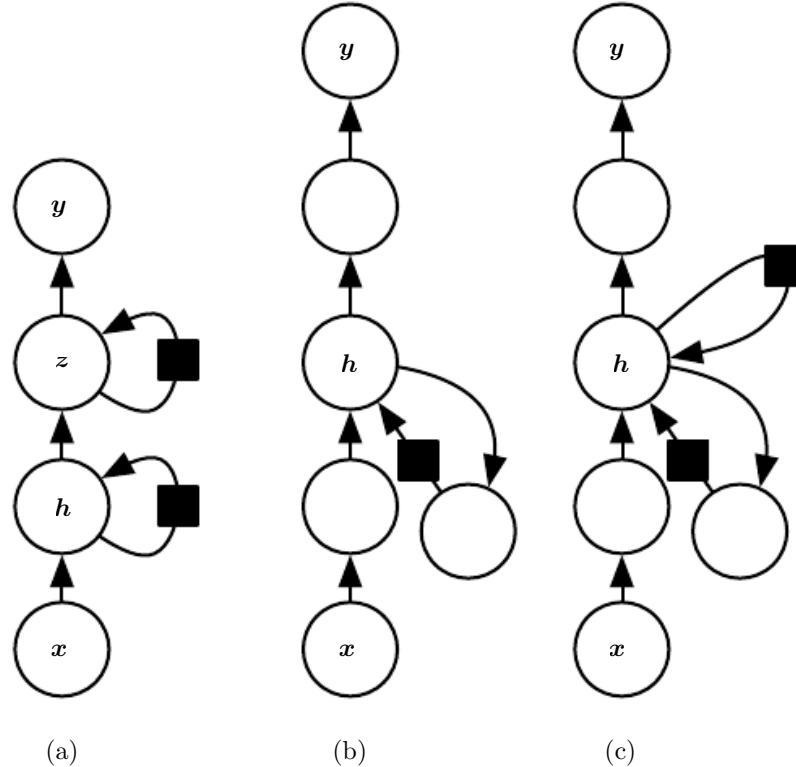


Figure 10.13: A recurrent neural network can be made deep in many ways (Pascanu *et al.*, 2014a). (a) The hidden recurrent state can be broken down into groups organized hierarchically. (b) Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts. This may lengthen the shortest path linking different time steps. (c) The path-lengthening effect can be mitigated by introducing skip connections.

2. from the previous hidden state to the next hidden state, and
3. from the hidden state to the output.

With the RNN architecture of figure 10.3, each of these three blocks is associated with a single weight matrix. In other words, when the network is unfolded, each of these blocks corresponds to a shallow transformation. By a shallow transformation, we mean a transformation that would be represented by a single layer within a deep MLP. Typically this is a transformation represented by a learned affine transformation followed by a fixed nonlinearity.

Would it be advantageous to introduce depth in each of these operations? Experimental evidence (Graves *et al.*, 2013; Pascanu *et al.*, 2014a) strongly suggests so. The experimental evidence is in agreement with the idea that we need enough

depth to perform the required mappings. See also Schmidhuber (1992), El Hihi and Bengio (1996), or Jaeger (2007a) for earlier work on deep RNNs.

Graves *et al.* (2013) were the first to show a significant benefit of decomposing the state of an RNN into multiple layers, as in figure 10.13 (left). We can think of the lower layers in the hierarchy depicted in figure 10.13a as playing a role in transforming the raw input into a representation that is more appropriate, at the higher levels of the hidden state. Pascanu *et al.* (2014a) go a step further and propose to have a separate MLP (possibly deep) for each of the three blocks enumerated above, as illustrated in figure 10.13b. Considerations of representational capacity suggest allocating enough capacity in each of these three steps, but doing so by adding depth may hurt learning by making optimization difficult. In general, it is easier to optimize shallower architectures, and adding the extra depth of figure 10.13b makes the shortest path from a variable in time step t to a variable in time step $t + 1$ become longer. For example, if an MLP with a single hidden layer is used for the state-to-state transition, we have doubled the length of the shortest path between variables in any two different time steps, compared with the ordinary RNN of figure 10.3. However, as argued by Pascanu *et al.* (2014a), this can be mitigated by introducing skip connections in the hidden-to-hidden path, as illustrated in figure 10.13c.

10.6 Recursive Neural Networks

Recursive neural networks² represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in figure 10.14. Recursive neural networks were introduced by Pollack (1990), and their potential use for learning to reason was described by Bottou (2011). Recursive networks have been successfully applied to processing *data structures* as input to neural nets (Frasconi *et al.*, 1997, 1998), in natural language processing (Socher *et al.*, 2011a,c, 2013a), as well as in computer vision (Socher *et al.*, 2011b).

One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ , the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to $O(\log \tau)$, which might help deal with long-term dependencies. An open question is how to best structure the tree. One option is to have a tree structure that does not depend on the data,

²We suggest not abbreviating “recursive neural network” as “RNN” to avoid confusion with “recurrent neural network.”

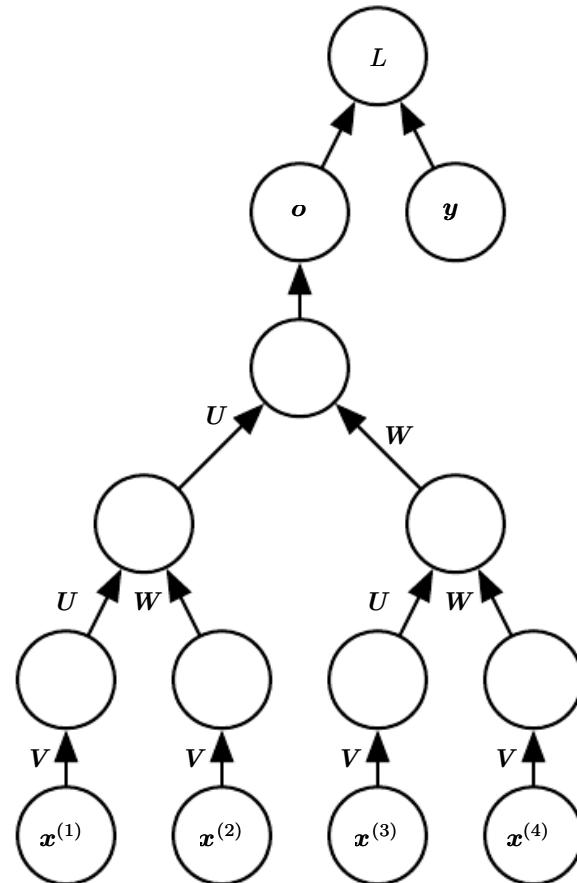


Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$ can be mapped to a fixed-size representation (the output \mathbf{o}), with a fixed set of parameters (the weight matrices $\mathbf{U}, \mathbf{V}, \mathbf{W}$). The figure illustrates a supervised learning case in which some target \mathbf{y} is provided that is associated with the whole sequence.

such as a balanced binary tree. In some application domains, external methods can suggest the appropriate tree structure. For example, when processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parser (Socher *et al.*, 2011a, 2013a). Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input, as suggested by Bottou (2011).

Many variants of the recursive net idea are possible. For example, Frasconi *et al.* (1997) and Frasconi *et al.* (1998) associate the data with a tree structure, and associate the inputs and targets with individual nodes of the tree. The computation performed by each node does not have to be the traditional artificial neuron computation (affine transformation of all inputs followed by a monotone nonlinearity). For example, Socher *et al.* (2013a) propose using tensor operations and bilinear forms, which have previously been found useful to model relationships between concepts (Weston *et al.*, 2010; Bordes *et al.*, 2012) when the concepts are represented by continuous vectors (embeddings).

10.7 The Challenge of Long-Term Dependencies

The mathematical challenge of learning long-term dependencies in recurrent networks is introduced in section 8.2.5. The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones. Many other sources provide a deeper treatment (Hochreiter, 1991; Doya, 1993; Bengio *et al.*, 1994; Pascanu *et al.*, 2013). In this section, we describe the problem in more detail. The remaining sections describe approaches to overcoming the problem.

Recurrent networks involve the composition of the same function multiple times, once per time step. These compositions can result in extremely nonlinear behavior, as illustrated in figure 10.15.

In particular, the function composition employed by recurrent neural networks somewhat resembles matrix multiplication. We can think of the recurrence relation

$$\mathbf{h}^{(t)} = \mathbf{W}^\top \mathbf{h}^{(t-1)} \quad (10.36)$$

as a very simple recurrent neural network lacking a nonlinear activation function,

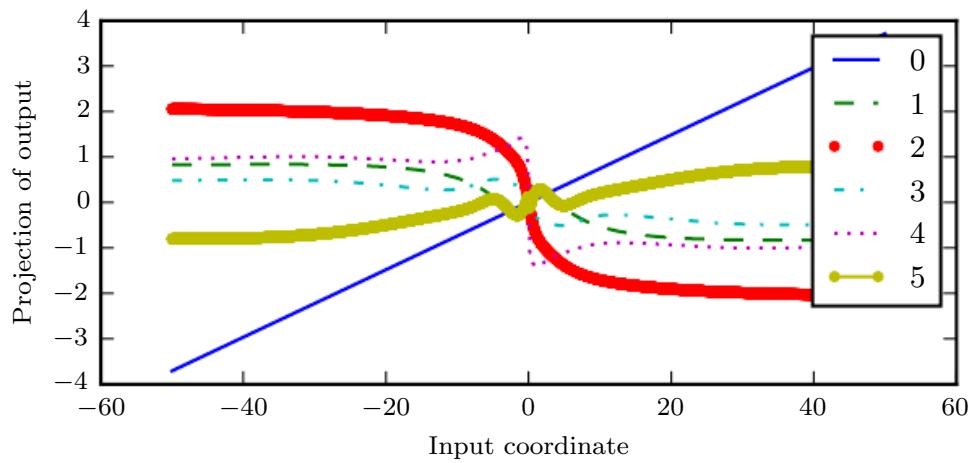


Figure 10.15: Repeated function composition. When composing many nonlinear functions (like the linear-tanh layer shown here), the result is highly nonlinear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many alternations between increasing and decreasing. Here, we plot a linear projection of a 100-dimensional hidden state down to a single dimension, plotted on the y -axis. The x -axis is the coordinate of the initial state along a random direction in the 100-dimensional space. We can thus view this plot as a linear cross-section of a high-dimensional function. The plots show the function after each time step, or equivalently, after each number of times the transition function has been composed.

and lacking inputs \mathbf{x} . As described in section 8.2.5, this recurrence relation essentially describes the power method. It may be simplified to

$$\mathbf{h}^{(t)} = (\mathbf{W}^t)^\top \mathbf{h}^{(0)}, \quad (10.37)$$

and if \mathbf{W} admits an eigendecomposition of the form

$$\mathbf{W} = \mathbf{Q}\Lambda\mathbf{Q}^\top, \quad (10.38)$$

with orthogonal \mathbf{Q} , the recurrence may be simplified further to

$$\mathbf{h}^{(t)} = \mathbf{Q}^\top \Lambda^t \mathbf{Q} \mathbf{h}^{(0)}. \quad (10.39)$$

The eigenvalues are raised to the power of t , causing eigenvalues with magnitude less than one to decay to zero and eigenvalues with magnitude greater than one to explode. Any component of $\mathbf{h}^{(0)}$ that is not aligned with the largest eigenvector will eventually be discarded.

This problem is particular to recurrent networks. In the scalar case, imagine multiplying a weight w by itself many times. The product w^t will either vanish or explode depending on the magnitude of w . If we make a nonrecurrent network that has a different weight $w^{(t)}$ at each time step, the situation is different. If the initial state is given by 1, then the state at time t is given by $\prod_t w^{(t)}$. Suppose that the $w^{(t)}$ values are generated randomly, independently from one another, with zero mean and variance v . The variance of the product is $O(v^n)$. To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$. Very deep feedforward networks with carefully chosen scaling can thus avoid the vanishing and exploding gradient problem, as argued by Sussillo (2014).

The vanishing and exploding gradient problem for RNNs was independently discovered by separate researchers (Hochreiter, 1991; Bengio *et al.*, 1993, 1994). One may hope that the problem can be avoided simply by staying in a region of parameter space where the gradients do not vanish or explode. Unfortunately, in order to store memories in a way that is robust to small perturbations, the RNN must enter a region of parameter space where gradients vanish (Bengio *et al.*, 1993, 1994). Specifically, whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction. This means not that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, the experiments in Bengio *et al.* (1994) show that as we increase the span of the dependencies that

need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20.

For a deeper treatment of recurrent networks as dynamical systems, see Doya (1993), Bengio *et al.* (1994), and Siegelmann and Sontag (1995), with a review in Pascanu *et al.* (2013). The remaining sections of this chapter discuss various approaches that have been proposed to reduce the difficulty of learning long-term dependencies (in some cases allowing an RNN to learn dependencies across hundreds of steps), but the problem of learning long-term dependencies remains one of the main challenges in deep learning.

10.8 Echo State Networks

The recurrent weights mapping from $\mathbf{h}^{(t-1)}$ to $\mathbf{h}^{(t)}$ and the input weights mapping from $\mathbf{x}^{(t)}$ to $\mathbf{h}^{(t)}$ are some of the most difficult parameters to learn in a recurrent network. One proposed (Jaeger, 2003; Maass *et al.*, 2002; Jaeger and Haas, 2004; Jaeger, 2007b) approach to avoiding this difficulty is to set the recurrent weights such that the recurrent hidden units do a good job of capturing the history of past inputs, and *only learn the output weights*. This is the idea that was independently proposed for **echo state networks**, or ESNs (Jaeger and Haas, 2004; Jaeger, 2007b), and **liquid state machines** (Maass *et al.*, 2002). The latter is similar, except that it uses spiking neurons (with binary outputs) instead of the continuous-valued hidden units used for ESNs. Both ESNs and liquid state machines are termed **reservoir computing** (Lukoševičius and Jaeger, 2009) to denote the fact that the hidden units form a reservoir of temporal features that may capture different aspects of the history of inputs.

One way to think about these reservoir computing recurrent networks is that they are similar to kernel machines: they map an arbitrary length sequence (the history of inputs up to time t) into a fixed-length vector (the recurrent state $\mathbf{h}^{(t)}$), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest. The training criterion may then be easily designed to be convex as a function of the output weights. For example, if the output consists of linear regression from the hidden units to the output targets, and the training criterion is mean squared error, then it is convex and may be solved reliably with simple learning algorithms (Jaeger, 2003).

The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.

The original idea was to make the eigenvalues of the Jacobian of the state-to-state transition function be close to 1. As explained in section 8.2.5, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $\mathbf{J}^{(t)} = \frac{\partial s^{(t)}}{\partial s^{(t-1)}}$. Of particular importance is the **spectral radius** of $\mathbf{J}^{(t)}$, defined to be the maximum of the absolute values of its eigenvalues.

To understand the effect of the spectral radius, consider the simple case of back-propagation with a Jacobian matrix \mathbf{J} that does not change with t . This case happens, for example, when the network is purely linear. Suppose that \mathbf{J} has an eigenvector \mathbf{v} with corresponding eigenvalue λ . Consider what happens as we propagate a gradient vector backward through time. If we begin with a gradient vector \mathbf{g} , then after one step of back-propagation, we will have $\mathbf{J}\mathbf{g}$, and after n steps we will have $\mathbf{J}^n\mathbf{g}$. Now consider what happens if we instead back-propagate a perturbed version of \mathbf{g} . If we begin with $\mathbf{g} + \delta\mathbf{v}$, then after one step, we will have $\mathbf{J}(\mathbf{g} + \delta\mathbf{v})$. After n steps, we will have $\mathbf{J}^n(\mathbf{g} + \delta\mathbf{v})$. From this we can see that back-propagation starting from \mathbf{g} and back-propagation starting from $\mathbf{g} + \delta\mathbf{v}$ diverge by $\delta\mathbf{J}^n\mathbf{v}$ after n steps of back-propagation. If \mathbf{v} is chosen to be a unit eigenvector of \mathbf{J} with eigenvalue λ , then multiplication by the Jacobian simply scales the difference at each step. The two executions of back-propagation are separated by a distance of $\delta|\lambda|^n$. When \mathbf{v} corresponds to the largest value of $|\lambda|$, this perturbation achieves the widest possible separation of an initial perturbation of size δ .

When $|\lambda| > 1$, the deviation size $\delta|\lambda|^n$ grows exponentially large. When $|\lambda| < 1$, the deviation size becomes exponentially small.

Of course, this example assumed that the Jacobian was the same at every time step, corresponding to a recurrent network with no nonlinearity. When a nonlinearity is present, the derivative of the nonlinearity will approach zero on many time steps and help prevent the explosion resulting from a large spectral radius. Indeed, the most recent work on echo state networks advocates using a spectral radius much larger than unity (Yildiz *et al.*, 2012; Jaeger, 2012).

Everything we have said about back-propagation via repeated matrix multiplication applies equally to forward propagation in a network with no nonlinearity, where the state $\mathbf{h}^{(t+1)} = \mathbf{h}^{(t)\top}\mathbf{W}$.

When a linear map \mathbf{W}^\top always shrinks \mathbf{h} as measured by the L^2 norm, then

we say that the map is **contractive**. When the spectral radius is less than one, the mapping from $\mathbf{h}^{(t)}$ to $\mathbf{h}^{(t+1)}$ is contractive, so a small change becomes smaller after each time step. This necessarily makes the network forget information about the past when we use a finite level of precision (such as 32-bit integers) to store the state vector.

The Jacobian matrix tells us how a small change of $\mathbf{h}^{(t)}$ propagates one step forward, or equivalently, how the gradient on $\mathbf{h}^{(t+1)}$ propagates one step backward, during back-propagation. Note that neither \mathbf{W} nor \mathbf{J} need to be symmetric (although they are square and real), so they can have complex-valued eigenvalues and eigenvectors, with imaginary components corresponding to potentially oscillatory behavior (if the same Jacobian was applied iteratively). Even though $\mathbf{h}^{(t)}$ or a small variation of $\mathbf{h}^{(t)}$ of interest in back-propagation are real valued, they can be expressed in such a complex-valued basis. What matters is what happens to the magnitude (complex absolute value) of these possibly complex-valued basis coefficients when we multiply the matrix by the vector. An eigenvalue with magnitude greater than one corresponds to magnification (exponential growth, if applied iteratively) while a magnitude smaller than one corresponds to shrinking (exponential decay, if applied iteratively).

With a nonlinear map, the Jacobian is free to change at each step. The dynamics therefore become more complicated. It remains true, however, that a small initial variation can turn into a large variation after several steps. One difference between the purely linear case and the nonlinear case is that the use of a squashing nonlinearity such as tanh can cause the recurrent dynamics to become bounded. Note that it is possible for back-propagation to retain unbounded dynamics even when forward propagation has bounded dynamics, for example, when a sequence of tanh units are all in the middle of their linear regime and are connected by weight matrices with spectral radius greater than 1. Nonetheless, it is rare for all the tanh units to simultaneously lie at their linear activation point.

The strategy of echo state networks is simply to fix the weights to have some spectral radius such as 3, where information is carried forward through time but does not explode because of the stabilizing effect of saturating nonlinearities like tanh.

More recently, it has been shown that the techniques used to set the weights in ESNs could be used to *initialize* the weights in a fully trainable recurrent network (with the hidden-to-hidden recurrent weights trained using back-propagation through time), helping to learn long-term dependencies (Sutskever, 2012; Sutskever *et al.*, 2013). In this setting, an initial spectral radius of 1.2 performs well, combined with the sparse initialization scheme described in section 8.4.

10.9 Leaky Units and Other Strategies for Multiple Time Scales

One way to deal with long-term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales and can handle small details, while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently. Various strategies for building both fine and coarse time scales are possible. These include the addition of skip connections across time, “leaky units” that integrate signals with different time constants, and the removal of some of the connections used to model fine-grained time scales.

10.9.1 Adding Skip Connections through Time

One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present. The idea of using such skip connections dates back to [Lin et al. \(1996\)](#) and follows from the idea of incorporating delays in feedforward neural networks ([Lang and Hinton, 1988](#)). In an ordinary recurrent network, a recurrent connection goes from a unit at time t to a unit at time $t + 1$. It is possible to construct recurrent networks with longer delays ([Bengio, 1991](#)).

As we have seen in section [8.2.5](#), gradients may vanish or explode exponentially *with respect to the number of time steps*. [Lin et al. \(1996\)](#) introduced recurrent connections with a time delay of d to mitigate this problem. Gradients now diminish exponentially as a function of $\frac{\tau}{d}$ rather than τ . Since there are both delayed and single step connections, gradients may still explode exponentially in τ . This allows the learning algorithm to capture longer dependencies, although not all long-term dependencies may be represented well in this way.

10.9.2 Leaky Units and a Spectrum of Different Time Scales

Another way to obtain paths on which the product of derivatives is close to one is to have units with *linear* self-connections and a weight near one on these connections.

When we accumulate a running average $\mu^{(t)}$ of some value $v^{(t)}$ by applying the update $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$, the α parameter is an example of a linear self-connection from $\mu^{(t-1)}$ to $\mu^{(t)}$. When α is near one, the running average remembers information about the past for a long time, and when α is near zero, information about the past is rapidly discarded. Hidden units with linear self-connections can behave similarly to such running averages. Such hidden units are called **leaky**

units.

Skip connections through d time steps are a way of ensuring that a unit can always learn to be influenced by a value from d time steps earlier. The use of a linear self-connection with a weight near one is a different way of ensuring that the unit can access values from the past. The linear self-connection approach allows this effect to be adapted more smoothly and flexibly by adjusting the real valued α rather than by adjusting the integer-valued skip length.

These ideas were proposed by Mozer (1992) and by El Hihi and Bengio (1996). Leaky units were also found to be useful in the context of echo state networks (Jaeger *et al.*, 2007).

There are two basic strategies for setting the time constants used by leaky units. One strategy is to manually fix them to values that remain constant, for example, by sampling their values from some distribution once at initialization time. Another strategy is to make the time constants free parameters and learn them. Having such leaky units at different time scales appears to help with long-term dependencies (Mozer, 1992; Pascanu *et al.*, 2013).

10.9.3 Removing Connections

Another approach to handling long-term dependencies is the idea of organizing the state of the RNN at multiple time scales (El Hihi and Bengio, 1996), with information flowing more easily through long distances at the slower time scales.

This idea differs from the skip connections through time discussed earlier because it involves actively *removing* length-one connections and replacing them with longer connections. Units modified in such a way are forced to operate on a long time scale. Skip connections through time *add* edges. Units receiving such new connections may learn to operate on a long time scale but may also choose to focus on their other, short-term connections.

There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky, but to have different groups of units associated with different fixed time scales. This was the proposal in Mozer (1992) and has been successfully used in Pascanu *et al.* (2013). Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units. This is the approach of El Hihi and Bengio (1996) and Koutnik *et al.* (2014). It worked well on a number of benchmark datasets.

10.10 The Long Short-Term Memory and Other Gated RNNs

As of this writing, the most effective sequence models used in practical applications are called **gated RNNs**. These include the **long short-term memory** and networks based on the **gated recurrent unit**.

Like leaky units, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. Leaky units did this with connection weights that were either manually chosen constants or were parameters. Gated RNNs generalize this to connection weights that may change at each time step.

Leaky units allow the network to *accumulate* information (such as evidence for a particular feature or category) over a long duration. Once that information has been used, however, it might be useful for the neural network to *forget* the old state. For example, if a sequence is made of subsequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it. This is what gated RNNs do.

10.10.1 LSTM

The clever idea of introducing self-loops to produce paths where the gradient can flow for long durations is a core contribution of the initial **long short-term memory** (LSTM) model (Hochreiter and Schmidhuber, 1997). A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed (Gers *et al.*, 2000). By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself. The LSTM has been found extremely successful in many applications, such as unconstrained handwriting recognition (Graves *et al.*, 2009), speech recognition (Graves *et al.*, 2013; Graves and Jaitly, 2014), handwriting generation (Graves, 2013), machine translation (Sutskever *et al.*, 2014), image captioning (Kiros *et al.*, 2014b; Vinyals *et al.*, 2014b; Xu *et al.*, 2015), and parsing (Vinyals *et al.*, 2014a).

The LSTM block diagram is illustrated in figure 10.16. The corresponding forward propagation equations are given below, for a shallow recurrent network

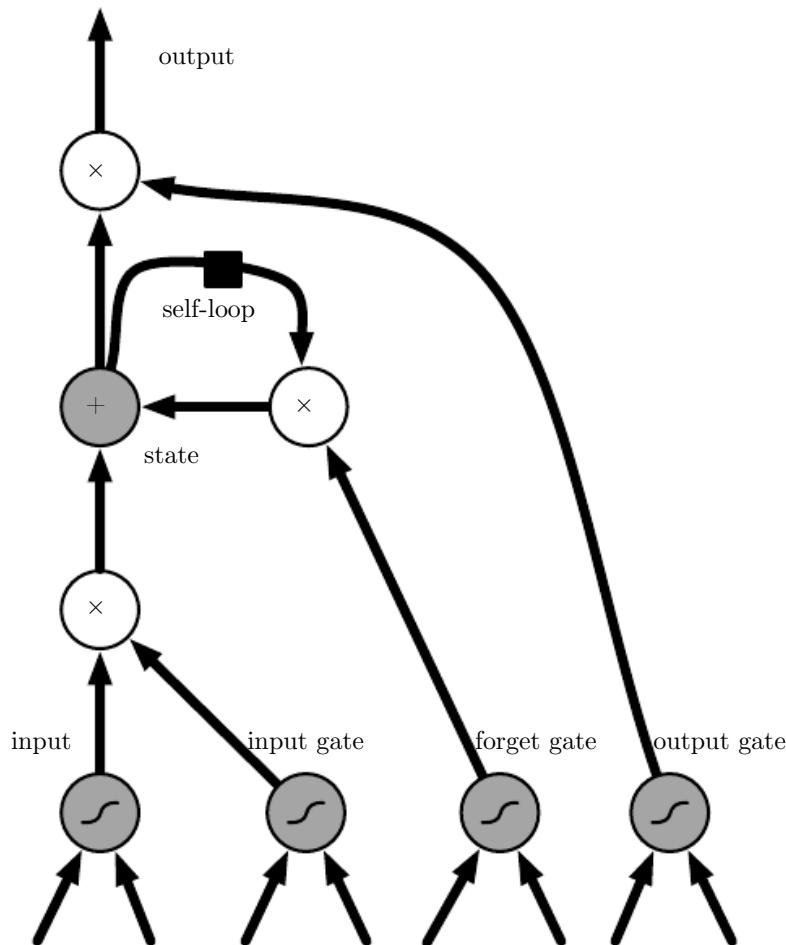


Figure 10.16: Block diagram of the LSTM recurrent network “cell.” Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

architecture. Deeper architectures have also been successfully used (Graves *et al.*, 2013; Pascanu *et al.*, 2014a). Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have “LSTM cells” that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop similar to the leaky units described in the previous section. Here, however, the self-loop weight (or the associated time constant) is controlled by a **forget gate** unit $f_i^{(t)}$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma \left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right), \quad (10.40)$$

where $\mathbf{x}^{(t)}$ is the current input vector and $\mathbf{h}^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and \mathbf{b}^f , \mathbf{U}^f , \mathbf{W}^f are respectively biases, input weights, and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right), \quad (10.41)$$

where \mathbf{b} , \mathbf{U} and \mathbf{W} respectively denote the biases, input weights, and recurrent weights into the LSTM cell. The **external input gate** unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma \left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right). \quad (10.42)$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the **output gate** $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)}, \quad (10.43)$$

$$q_i^{(t)} = \sigma \left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right), \quad (10.44)$$

which has parameters \mathbf{b}^o , \mathbf{U}^o , \mathbf{W}^o for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i -th unit, as shown in figure 10.16. This would require three additional parameters.

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial datasets designed for testing the ability to learn long-term dependencies (Bengio *et al.*, 1994; Hochreiter and Schmidhuber, 1997; Hochreiter *et al.*, 2001), then on challenging sequence processing tasks where state-of-the-art performance was obtained (Graves, 2012; Graves *et al.*, 2013; Sutskever *et al.*, 2014). Variants and alternatives to the LSTM that have been studied and used are discussed next.

10.10.2 Other Gated RNNs

Which pieces of the LSTM architecture are actually necessary? What other successful architectures could be designed that allow the network to dynamically control the time scale and forgetting behavior of different units?

Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as gated recurrent units, or GRUs (Cho *et al.*, 2014b; Chung *et al.*, 2014, 2015a; Jozefowicz *et al.*, 2015; Chrupala *et al.*, 2015). The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit. The update equations are the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right), \quad (10.45)$$

where \mathbf{u} stands for “update” gate and \mathbf{r} for “reset” gate. Their value is defined as usual:

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right) \quad (10.46)$$

and

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right). \quad (10.47)$$

The reset and update gates can individually “ignore” parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any

dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it with the new “target state” value (toward which the leaky integrator wants to converge). The reset gates control which parts of the state get used to compute the next target state, introducing an additional nonlinear effect in the relationship between past state and future state.

Many more variants around this theme can be designed. For example the reset gate (or forget gate) output could be shared across multiple hidden units. Alternately, the product of a global gate (covering a whole group of units, such as an entire layer) and a local gate (per unit) could be used to combine global control and local control. Several investigations over architectural variations of the LSTM and GRU, however, found no variant that would clearly beat both of these across a wide range of tasks (Greff *et al.*, 2015; Jozefowicz *et al.*, 2015). Greff *et al.* (2015) found that a crucial ingredient is the forget gate, while Jozefowicz *et al.* (2015) found that adding a bias of 1 to the LSTM forget gate, a practice advocated by Gers *et al.* (2000), makes the LSTM as strong as the best of the explored architectural variants.

10.11 Optimization for Long-Term Dependencies

Section 8.2.5 and section 10.7 have described the vanishing and exploding gradient problems that occur when optimizing RNNs over many time steps.

An interesting idea proposed by Martens and Sutskever (2011) is that second derivatives may vanish at the same time that first derivatives vanish. Second-order optimization algorithms may roughly be understood as dividing the first derivative by the second derivative (in higher dimension, multiplying the gradient by the inverse Hessian). If the second derivative shrinks at a similar rate to the first derivative, then the ratio of first and second derivatives may remain relatively constant. Unfortunately, second-order methods have many drawbacks, including high computational cost, the need for a large minibatch, and a tendency to be attracted to saddle points. Martens and Sutskever (2011) found promising results using second-order methods. Later, Sutskever *et al.* (2013) found that simpler methods such as Nesterov momentum with careful initialization could achieve similar results. See Sutskever (2012) for more detail. Both of these approaches have largely been replaced by simply using SGD (even without momentum) applied to LSTMs. This is part of a continuing theme in machine learning that it is often much easier to design a model that is easy to optimize than it is to design a more powerful optimization algorithm.

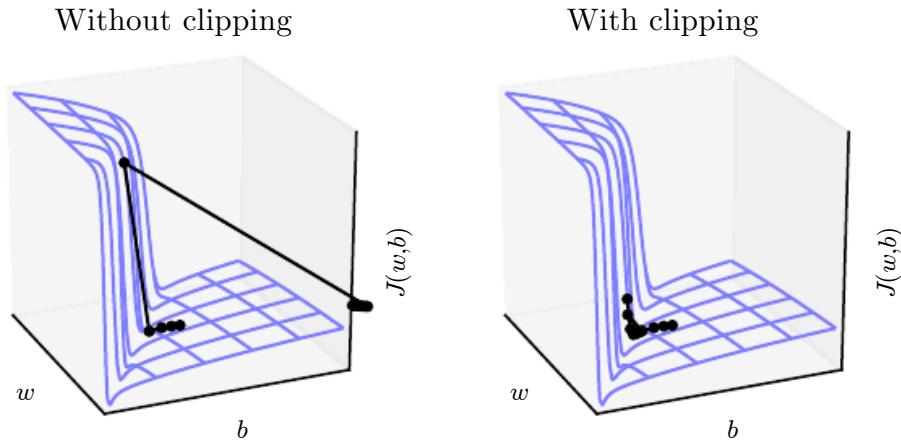


Figure 10.17: Example of the effect of gradient clipping in a recurrent network with two parameters \mathbf{w} and \mathbf{b} . Gradient clipping can make gradient descent perform more reasonably in the vicinity of extremely steep cliffs. These steep cliffs commonly occur in recurrent networks near where a recurrent network behaves approximately linearly. The cliff is exponentially steep in the number of time steps because the weight matrix is multiplied by itself once for each time step. (*Left*) Gradient descent without gradient clipping overshoots the bottom of this small ravine, then receives a very large gradient from the cliff face. The large gradient catastrophically propels the parameters outside the axes of the plot. (*Right*) Gradient descent with gradient clipping has a more moderate reaction to the cliff. While it does ascend the cliff face, the step size is restricted so that it cannot be propelled away from the steep region near the solution. Figure adapted with permission from [Pascanu et al. \(2013\)](#).

10.11.1 Clipping Gradients

As discussed in section 8.2.4, strongly nonlinear functions, such as those computed by a recurrent net over many time steps, tend to have derivatives that can be either very large or very small in magnitude. This is illustrated in figure 8.3 and in figure 10.17, in which we see that the objective function (as a function of the parameters) has a “landscape” in which one finds “cliffs”: wide and rather flat regions separated by tiny regions where the objective function changes quickly, forming a kind of cliff.

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution. The gradient tells us the direction that corresponds to the steepest descent within an infinitesimal region surrounding the current parameters. Outside this infinitesimal region, the cost function may

begin to curve back upward. The update must be chosen to be small enough to avoid traversing too much upward curvature. We typically use learning rates that decay slowly enough that consecutive steps have approximately the same learning rate. A step size that is appropriate for a relatively linear part of the landscape is often inappropriate and causes uphill motion if we enter a more curved part of the landscape on the next step.

A simple type of solution has been in use by practitioners for many years: **clipping the gradient**. There are different instances of this idea ([Mikolov, 2012](#); [Pascanu et al., 2013](#)). One option is to clip the parameter gradient from a minibatch *element-wise* ([Mikolov, 2012](#)), just before the parameter update. Another is to *clip the norm* $\|\mathbf{g}\|$ of the gradient \mathbf{g} ([Pascanu et al., 2013](#)) just before the parameter update:

$$\text{if } \|\mathbf{g}\| > v \quad (10.48)$$

$$\mathbf{g} \leftarrow \frac{\mathbf{g}v}{\|\mathbf{g}\|}, \quad (10.49)$$

where v is the norm threshold and \mathbf{g} is used to update parameters. Because the gradient of all the parameters (including different groups of parameters, such as weights and biases) is renormalized jointly with a single scaling factor, the latter method has the advantage of guaranteeing that each step is still in the gradient direction, but experiments suggest that both forms work similarly. Although the parameter update has the same direction as the true gradient, with gradient norm clipping, the parameter update vector norm is now bounded. This bounded gradient avoids performing a detrimental step when the gradient explodes. In fact, even simply taking a *random step* when the gradient magnitude is above a threshold tends to work almost as well. If the explosion is so severe that the gradient is numerically `Inf` or `Nan` (considered infinite or not-a-number), then a random step of size v can be taken and will typically move away from the numerically unstable configuration. Clipping the gradient norm per minibatch will not change the direction of the gradient for an individual minibatch. Taking the average of the norm-clipped gradient from many minibatches, however, is not equivalent to clipping the norm of the true gradient (the gradient formed from using all examples). Examples that have large gradient norm, as well as examples that appear in the same minibatch as such examples, will have their contribution to the final direction diminished. This stands in contrast to traditional minibatch gradient descent, where the true gradient direction is equal to the average over all minibatch gradients. Put another way, traditional stochastic gradient descent uses an unbiased estimate of the gradient, while gradient descent with norm clipping introduces a heuristic bias that we know empirically to be useful. With element-

wise clipping, the direction of the update is not aligned with the true gradient or the minibatch gradient, but it is still a descent direction. It has also been proposed (Graves, 2013) to clip the back-propagated gradient (with respect to hidden units), but no comparison has been published between these variants; we conjecture that all these methods behave similarly.

10.11.2 Regularizing to Encourage Information Flow

Gradient clipping helps to deal with exploding gradients, but it does not help with vanishing gradients. To address vanishing gradients and better capture long-term dependencies, we discussed the idea of creating paths in the computational graph of the unfolded recurrent architecture along which the product of gradients associated with arcs is near 1. One approach to achieve this is with LSTMs and other self-loops and gating mechanisms, described in section 10.10. Another idea is to regularize or constrain the parameters so as to encourage “information flow.” In particular, we would like the gradient vector $\nabla_{\mathbf{h}^{(t)}} L$ being back-propagated to maintain its magnitude, even if the loss function only penalizes the output at the end of the sequence. Formally, we want

$$(\nabla_{\mathbf{h}^{(t)}} L) \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \quad (10.50)$$

to be as large as

$$\nabla_{\mathbf{h}^{(t)}} L. \quad (10.51)$$

With this objective, Pascanu *et al.* (2013) propose the following regularizer:

$$\Omega = \sum_t \left(\frac{\left\| (\nabla_{\mathbf{h}^{(t)}} L) \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \right\|}{\|\nabla_{\mathbf{h}^{(t)}} L\|} - 1 \right)^2. \quad (10.52)$$

Computing the gradient of this regularizer may appear difficult, but Pascanu *et al.* (2013) propose an approximation in which we consider the back-propagated vectors $\nabla_{\mathbf{h}^{(t)}} L$ as if they were constants (for the purpose of this regularizer, so that there is no need to back-propagate through them). The experiments with this regularizer suggest that, if combined with the norm clipping heuristic (which handles gradient explosion), the regularizer can considerably increase the span of the dependencies that an RNN can learn. Because it keeps the RNN dynamics on the edge of explosive gradients, the gradient clipping is particularly important. Without gradient clipping, gradient explosion prevents learning from succeeding.

A key weakness of this approach is that it is not as effective as the LSTM for tasks where data is abundant, such as language modeling.

10.12 Explicit Memory

Intelligence requires knowledge, and acquiring knowledge can be done via learning, which has motivated the development of large-scale deep architectures. However, there are different kinds of knowledge. Some knowledge can be implicit, subconscious, and difficult to verbalize—such as how to walk, or how a dog looks different from a cat. Other knowledge can be explicit, declarative, and relatively straightforward to put into words—everyday commonsense knowledge, like “a cat is a kind of animal,” or very specific facts that you need to know to accomplish your current goals, like “the meeting with the sales team is at 3:00 PM in room 141.”

Neural networks excel at storing implicit knowledge, but they struggle to memorize facts. Stochastic gradient descent requires many presentations of the same input before it can be stored in neural network parameters, and even then, that input will not be stored especially precisely. Graves *et al.* (2014) hypothesized that this is because neural networks lack the equivalent of the **working memory** system that enables human beings to explicitly hold and manipulate pieces of information that are relevant to achieving some goal. Such explicit memory components would allow our systems not only to rapidly and “intentionally” store and retrieve specific facts but also to sequentially reason with them. The need for neural networks that can process information in a sequence of steps, changing the way the input is fed into the network at each step, has long been recognized as important for the ability to reason rather than to make automatic, intuitive responses to the input (Hinton, 1990).

To resolve this difficulty, Weston *et al.* (2014) introduced **memory networks** that include a set of memory cells that can be accessed via an addressing mechanism. Memory networks originally required a supervision signal instructing them how to use their memory cells. Graves *et al.* (2014) introduced the **neural Turing machine**, which is able to learn to read from and write arbitrary content to memory cells without explicit supervision about which actions to undertake, and allowed end-to-end training without this supervision signal, via the use of a content-based soft attention mechanism (see Bahdanau *et al.* [2015] and section 12.4.5.1). This soft addressing mechanism has become standard with other related architectures, emulating algorithmic mechanisms in a way that still allows gradient-based optimization (Sukhbaatar *et al.*, 2015; Joulin and Mikolov, 2015; Kumar *et al.*, 2015; Vinyals *et al.*, 2015a; Grefenstette *et al.*, 2015).

Each memory cell can be thought of as an extension of the memory cells in LSTMs and GRUs. The difference is that the network outputs an internal state that chooses which cell to read from or write to, just as memory accesses in a

digital computer read from or write to a specific address.

It is difficult to optimize functions that produce exact integer addresses. To alleviate this problem, NTMs actually read to or write from many memory cells simultaneously. To read, they take a weighted average of many cells. To write, they modify multiple cells by different amounts. The coefficients for these operations are chosen to be focused on a small number of cells, for example, by producing them via a softmax function. Using these weights with nonzero derivatives enables the functions controlling access to the memory to be optimized using gradient descent. The gradient on these coefficients indicates whether each of them should be increased or decreased, but the gradient will typically be large only for those memory addresses receiving a large coefficient.

These memory cells are typically augmented to contain a vector, rather than the single scalar stored by an LSTM or GRU memory cell. There are two reasons to increase the size of the memory cell. One reason is that we have increased the cost of accessing a memory cell. We pay the computational cost of producing a coefficient for many cells, but we expect these coefficients to cluster around a small number of cells. By reading a vector value, rather than a scalar value, we can offset some of this cost. Another reason to use vector valued memory cells is that they allow for **content-based addressing**, where the weight used to read to or write from a cell is a function of that cell. Vector valued cells allow us to retrieve a complete vector valued memory if we are able to produce a pattern that matches some but not all its elements. This is analogous to how people can recall the lyrics of a song based on a few words. We can think of a content-based read instruction as saying, “Retrieve the lyrics of the song that has the chorus ‘We all live in a yellow submarine.’” Content-based addressing is more useful when we make the objects to be retrieved large—if every letter of the song was stored in a separate memory cell, we would not be able to find them this way. By comparison, **location-based addressing** is not allowed to refer to the content of the memory. We can think of a location-based read instruction as saying “Retrieve the lyrics of the song in slot 347.” Location-based addressing can often be a perfectly sensible mechanism even when the memory cells are small.

If the content of a memory cell is copied (not forgotten) at most time steps, then the information it contains can be propagated forward in time and the gradients propagated backward in time without either vanishing or exploding.

The explicit memory approach is illustrated in figure 10.18, where we see that a “task neural network” is coupled with a memory. Although that task neural network could be feedforward or recurrent, the overall system is a recurrent network. The task network can choose to read from or write to specific memory addresses.

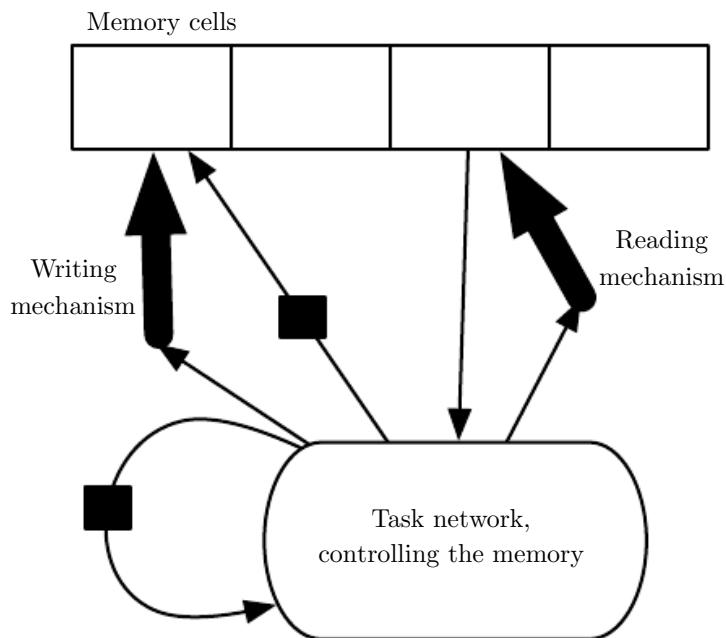


Figure 10.18: A schematic of a network with an explicit memory, capturing some of the key design elements of the neural Turing machine. In this diagram we distinguish the “representation” part of the model (the “task network,” here a recurrent net in the bottom) from the “memory” part of the model (the set of cells), which can store facts. The task network learns to “control” the memory, deciding where to read from and where to write to within the memory (through the reading and writing mechanisms, indicated by bold arrows pointing at the reading and writing addresses).

Explicit memory seems to allow models to learn tasks that ordinary RNNs or LSTM RNNs cannot learn. One reason for this advantage may be that information and gradients can be propagated (forward in time or backward in time, respectively) for very long durations.

As an alternative to back-propagation through weighted averages of memory cells, we can interpret the memory addressing coefficients as probabilities and stochastically read just one cell (Zaremba and Sutskever, 2015). Optimizing models that make discrete decisions requires specialized optimization algorithms, described in section 20.9.1. So far, training these stochastic architectures that make discrete decisions remains harder than training deterministic algorithms that make soft decisions.

Whether it is soft (allowing back-propagation) or stochastic and hard, the mechanism for choosing an address is in its form identical to the **attention mechanism**, which had been previously introduced in the context of machine translation (Bahdanau *et al.*, 2015) and is discussed in section 12.4.5.1. The idea of attention mechanisms for neural networks was introduced even earlier, in the context of handwriting generation (Graves, 2013), with an attention mechanism that was constrained to move only forward in time through the sequence. In the case of machine translation and memory networks, at each step, the focus of attention can move to a completely different place, compared to the previous step.

Recurrent neural networks provide a way to extend deep learning to sequential data. They are the last major tool in our deep learning toolbox. Our discussion now moves to how to choose and use these tools and how to apply them to real-world tasks.