Deep Learning

Ian Goodfellow
Yoshua Bengio and
Aaron Courville

The MIT Press
Cambridge, Massachusetts
London, England

Microsoft Exhibit 1043 - Page 1
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

© 2016 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set in SFRM1095 by diacriTech, Chennai.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Goodfellow, Ian, author. | Bengio, Yoshua, author. | Courville, Aaron, author.
Title: Deep learning / Ian Goodfellow, Yoshua Bengio, and Aaron Courville.
Description: Cambridge, MA : MIT Press, [2017] | Series: Adaptive computation and machine learning series | Includes bibliographical references and index.
Identifiers: LCCN 2016022992 | ISBN 9780262035613 (hardcover : alk. paper)
Subjects: LCSH: Machine learning,
Classification: LCC Q325.5 .G66 2017 | DDC 006.3/1–dc23 LC record available at https://lccn.loc.gov/2016022992

10 9 8 7

Microsoft Exhibit 1043 - Page 2
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 3
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

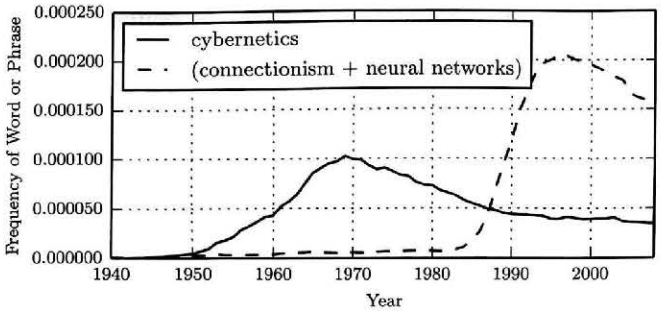


Figure 1.7: Two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases "cybernetics" and "connectionism" or "neural networks," according to Google Books (the third wave is too recent to appear). The first wave started with cybernetics in the 1940s–1960s, 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.

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.

Microsoft Exhibit 1043 - Page 4
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $x_1, \ldots, x_n$ and associate them with an output y. These models would learn a set of weights $w_1, \ldots, w_n$ and compute their output $f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + \cdots + x_n w_n$. 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 $f(\boldsymbol{x}, \boldsymbol{w})$ 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 $f(\boldsymbol{x})$ 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 $f(\boldsymbol{x}, \boldsymbol{w})$ 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 $f([0, 1], \boldsymbol{w}) = 1$ and $f([1, 0], \boldsymbol{w}) = 1$ but $f([1, 1], \boldsymbol{w}) = 0$ and $f([0, 0], \boldsymbol{w}) = 0$. 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.

Microsoft Exhibit 1043 - Page 5
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 6
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 7
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 was also included neuroscientists and experts in human and computer vision.

Microsoft Exhibit 1043 - Page 8
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 9
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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" has made machine learning much easier because the key burden of statistical estimation—generalizing well to new data after observing only a small amount

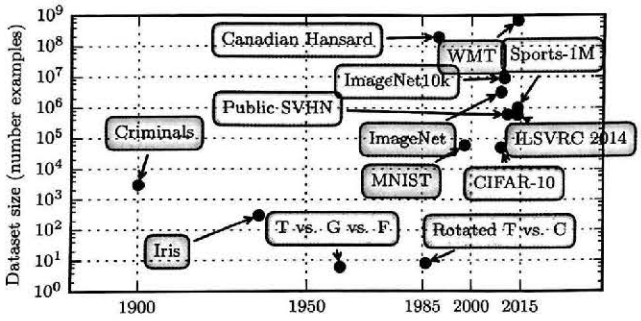


Figure 1.8: Increasing dataset size over time. In the early 1900s, statisticians studied datasets using hundreds or thousands of manually compiled measurements (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). In the 1950s through the 1980s, the pioneers of biologically inspired machine learning 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.

Microsoft Exhibit 1043 - Page 10
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

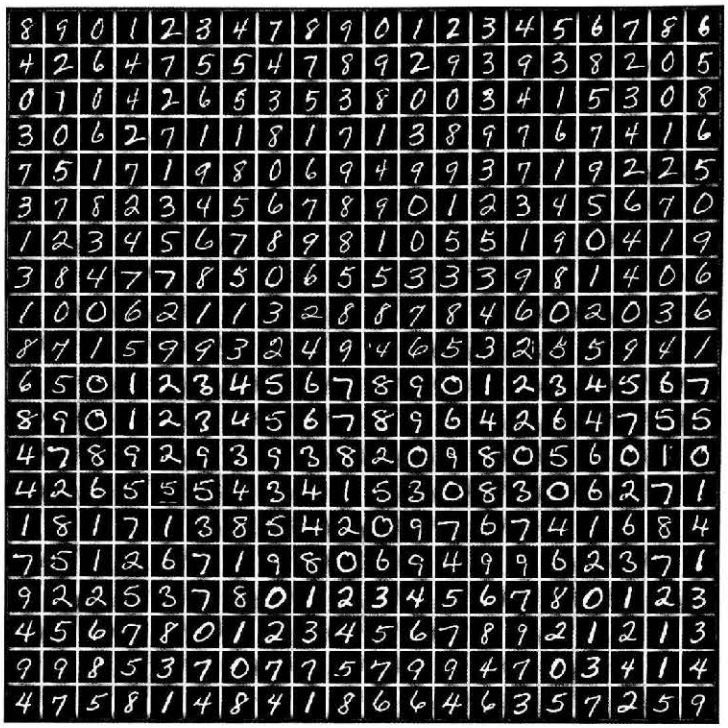


Figure 1.9: Example inputs from the MNIST dataset. The "NIST" stands for National Institute of Standards and Technology, the agency that originally collected this data. The "M" stands for "modified," since the data has been preprocessed for easier use with machine learning algorithms. The MNIST dataset 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.

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

Microsoft Exhibit 1043 - Page 11
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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. Increasing neural network size over time. 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.

Microsoft Exhibit 1043 - Page 12
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

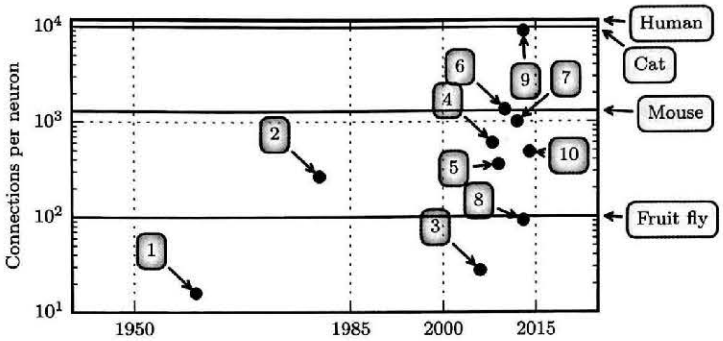


Figure 1.10: Number of connections per neuron over time. Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly 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 have a requirement that the photo be cropped near the object to be recognized

Microsoft Exhibit 1043 - Page 13
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

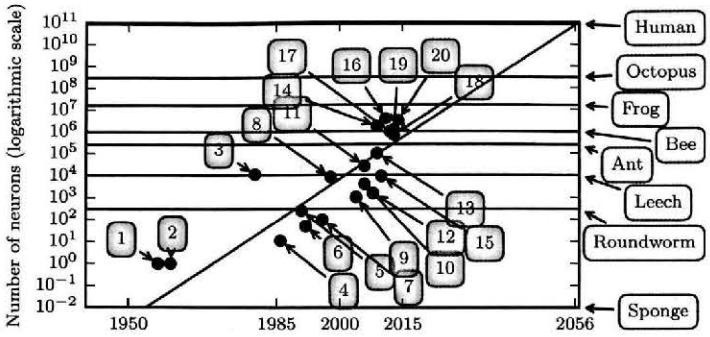


Figure 1.11: Increasing neural network size over time. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

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)

(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

Microsoft Exhibit 1043 - Page 14
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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, 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,

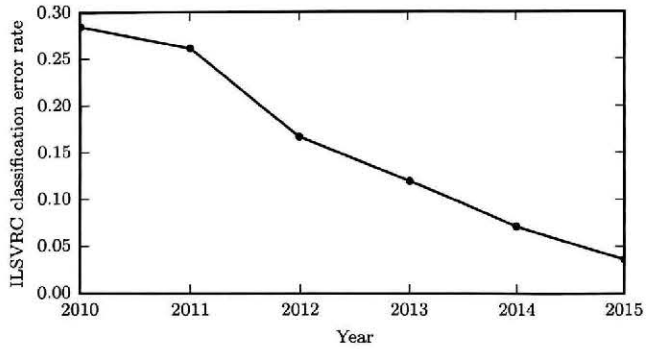


Figure 1.12: Decreasing error rate over time. Since deep networks reached the scale necessary to compete in the ImageNet Large Scale Visual Recognition Challenge, they have consistently won the competition every year, yielding lower and lower error rates each time. Data from Russakovsky et al. (2014b)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

Microsoft Exhibit 1043 - Page 15
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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., 2014a) 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

Microsoft Exhibit 1043 - Page 16
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

Microsoft Exhibit 1043 - Page 17
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

or contract space, the infinitesimal volume surrounding $\boldsymbol{x}$ in $\boldsymbol{x}$ space may have different volume in $\boldsymbol{y}$ space.

To see how to correct the problem, we return to the scalar case. We need to preserve the property

$$|p_y(g(x))dy| = |p_x(x)dx|. \tag{3.44}$$

Solving from this, we obtain

$$p_y(y) = p_x(g^{-1}(y)) \left| \frac{\partial x}{\partial y} \right| \tag{3.45}$$

or equivalently

$$p_x(x) = p_y(g(x)) \left| \frac{\partial g(x)}{\partial x} \right| . \tag{3.46}$$

In higher dimensions, the derivative generalizes to the determinant of the **Jacobian matrix**—the matrix with $J_{i,j} = \frac{\partial x_i}{\partial y_j}$. Thus, for real-valued vectors $\boldsymbol{x}$ and $\boldsymbol{y}$,

$$p_x(\boldsymbol{x}) = p_y(g(\boldsymbol{x})) \left| \det \left(\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}} \right) \right| . \tag{3.47}$$

3.13 Information Theory

Information theory is a branch of applied mathematics that revolves around quantifying how much information is present in a signal. It was originally invented to study sending messages from discrete alphabets over a noisy channel, such as communication via radio transmission. In this context, information theory tells how to design optimal codes and calculate the expected length of messages sampled from specific probability distributions using various encoding schemes. In the context of machine learning, we can also apply information theory to continuous variables where some of these message length interpretations do not apply. This field is fundamental to many areas of electrical engineering and computer science. In this textbook, we mostly use a few key ideas from information theory to characterize probability distributions or to quantify similarity between probability distributions. For more detail on information theory, see Cover and Thomas (2006) or MacKay (2003).

The basic intuition behind information theory is that learning that an unlikely event has occurred is more informative than learning that a likely event has occurred. A message saying "the sun rose this morning" is so uninformative as to be unnecessary to send, but a message saying "there was a solar eclipse this morning" is very informative.

Microsoft Exhibit 1043 - Page 18
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

We would like to quantify information in a way that formalizes this intuition.

- Likely events should have low information content, and in the extreme case, events that are guaranteed to happen should have no information content whatsoever.
- Less likely events should have higher information content.
- Independent events should have additive information. For example, finding out that a tossed coin has come up as heads twice should convey twice as much information as finding out that a tossed coin has come up as heads once.

To satisfy all three of these properties, we define the **self-information** of an event $\mathrm{x} = x$ to be

$$I(x) = -\log P(x). \tag{3.48}$$

In this book, we always use log to mean the natural logarithm, with base e. Our definition of $I(x)$ is therefore written in units of **nats**. One nat is the amount of information gained by observing an event of probability $\frac{1}{e}$. Other texts use base-2 logarithms and units called **bits** or **shannons**; information measured in bits is just a rescaling of information measured in nats.

When x is continuous, we use the same definition of information by analogy, but some of the properties from the discrete case are lost. For example, an event with unit density still has zero information, despite not being an event that is guaranteed to occur.

Self-information deals only with a single outcome. We can quantify the amount of uncertainty in an entire probability distribution using the **Shannon entropy**,

$$H(\mathrm{x}) = \mathbb{E}_{\mathrm{x}\sim P}[I(x)] = -\mathbb{E}_{\mathrm{x}\sim P}[\log P(x)], \tag{3.49}$$

also denoted $H(P)$. In other words, the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution. It gives a lower bound on the number of bits (if the logarithm is base 2, otherwise the units are different) needed on average to encode symbols drawn from a distribution P. Distributions that are nearly deterministic (where the outcome is nearly certain) have low entropy; distributions that are closer to uniform have high entropy. See figure 3.5 for a demonstration. When x is continuous, the Shannon entropy is known as the **differential entropy**.

If we have two separate probability distributions $P(\mathrm{x})$ and $Q(\mathrm{x})$ over the same random variable x, we can measure how different these two distributions are using

Microsoft Exhibit 1043 - Page 19
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

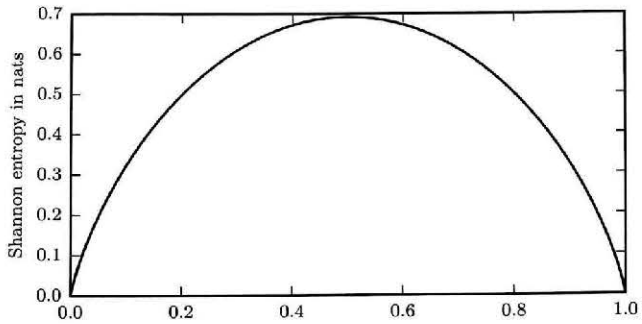


Figure 3.5: Shannon entropy of a binary random variable. This plot shows how distributions that are closer to deterministic have low Shannon entropy while distributions that are close to uniform have high Shannon entropy. On the horizontal axis, we plot p, the probability of a binary random variable being equal to 1. The entropy is given by $(p-1)\log(1-p)-p\log p$. When p is near 0, the distribution is nearly deterministic, because the random variable is nearly always 0. When p is near 1, the distribution is nearly deterministic, because the random variable is nearly always 1. When $p = 0.5$, the entropy is maximal, because the distribution is uniform over the two outcomes.

the **Kullback-Leibler (KL) divergence**:

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{\mathrm{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathrm{x}\sim P}\left[\log P(x) - \log Q(x)\right]. \tag{3.50}$$

In the case of discrete variables, it is the extra amount of information (measured in bits if we use the base-2 logarithm, but in machine learning we usually use nats and the natural logarithm) needed to send a message containing symbols drawn from probability distribution P, when we use a code that was designed to minimize the length of messages drawn from probability distribution Q.

The KL divergence has many useful properties, most notably being non-negative. The KL divergence is 0 if and only if P and Q are the same distribution in the case of discrete variables, or equal "almost everywhere" in the case of continuous variables. Because the KL divergence is non-negative and measures the difference between two distributions, it is often conceptualized as measuring some sort of distance between these distributions. It is not a true distance measure because it is not symmetric: $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$ for some P and Q. This asymmetry means that there are important consequences to the choice of whether to use $D_{\mathrm{KL}}(P\|Q)$ or $D_{\mathrm{KL}}(Q\|P)$. See figure 3.6 for more detail.

Microsoft Exhibit 1043 - Page 20
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

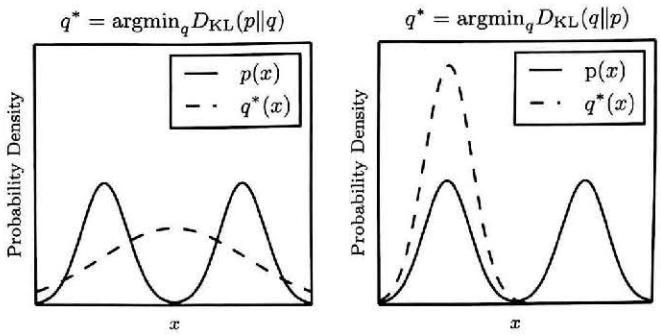


Figure 3.6: The KL divergence is asymmetric. Suppose we have a distribution $p(x)$ and wish to approximate it with another distribution $q(x)$. We have the choice of minimizing either $D_{\text{KL}}(p\|q)$ or $D_{\text{KL}}(q\|p)$. We illustrate the effect of this choice using a mixture of two Gaussians for p, and a single Gaussian for q. The choice of which direction of the KL divergence to use is problem dependent. Some applications require an approximation that usually places high probability anywhere that the true distribution places high probability, while other applications require an approximation that rarely places high probability anywhere that the true distribution places low probability. The choice of the direction of the KL divergence reflects which of these considerations takes priority for each application. *(Left)*The effect of minimizing $D_{\text{KL}}(p\|q)$. In this case, we select a q that has high probability where p has high probability. When p has multiple modes, q chooses to blur the modes together, in order to put high probability mass on all of them. *(Right)*The effect of minimizing $D_{\text{KL}}(q\|p)$. In this case, we select a q that has low probability where p has low probability. When p has multiple modes that are sufficiently widely separated, as in this figure, the KL divergence is minimized by choosing a single mode, to avoid putting probability mass in the low-probability areas between modes of p. Here, we illustrate the outcome when q is chosen to emphasize the left mode. We could also have achieved an equal value of the KL divergence by choosing the right mode. If the modes are not separated by a sufficiently strong low-probability region, then this direction of the KL divergence can still choose to blur the modes.

A quantity that is closely related to the KL divergence is the **cross-entropy** $H(P,Q) = H(P) + D_{\text{KL}}(P\|Q)$, which is similar to the KL divergence but lacking the term on the left:

$$H(P,Q) = -\mathbb{E}_{\mathbf{x}\sim P} \log Q(x). \tag{3.51}$$

Minimizing the cross-entropy with respect to Q is equivalent to minimizing the KL divergence, because Q does not participate in the omitted term.

When computing many of these quantities, it is common to encounter expressions of the form $0 \log 0$. By convention, in the context of information theory, we treat these expressions as $\lim_{x\to 0} x \log x = 0$.

Microsoft Exhibit 1043 - Page 21
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

3.14 Structured Probabilistic Models

Machine learning algorithms often involve probability distributions over a very large number of random variables. Often, these probability distributions involve direct interactions between relatively few variables. Using a single function to describe the entire joint probability distribution can be very inefficient (both computationally and statistically).

Instead of using a single function to represent a probability distribution, we can split a probability distribution into many factors that we multiply together. For example, suppose we have three random variables: a, b and c. Suppose that a influences the value of b, and b influences the value of c, but that a and c are independent given b. We can represent the probability distribution over all three variables as a product of probability distributions over two variables:

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{b}). \tag{3.52}$$

These factorizations can greatly reduce the number of parameters needed to describe the distribution. Each factor uses a number of parameters that is exponential in the number of variables in the factor. This means that we can greatly reduce the cost of representing a distribution if we are able to find a factorization into distributions over fewer variables.

We can describe these kinds of factorizations using graphs. Here, we use the word "graph" in the sense of graph theory: a set of vertices that may be connected to each other with edges. When we represent the factorization of a probability distribution with a graph, we call it a **structured probabilistic model**, or **graphical model**.

There are two main kinds of structured probabilistic models: directed and undirected. Both kinds of graphical models use a graph $\mathcal{G}$ in which each node in the graph corresponds to a random variable, and an edge connecting two random variables means that the probability distribution is able to represent direct interactions between those two random variables.

Directed models use graphs with directed edges, and they represent factorizations into conditional probability distributions, as in the example above. Specifically, a directed model contains one factor for every random variable x_i in the distribution, and that factor consists of the conditional distribution over x_i given the parents of x_i, denoted $Pa_{\mathcal{G}}(\mathrm{x}_i)$:

$$p(\mathbf{x}) = \prod_i p\left(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i)\right). \tag{3.53}$$

Microsoft Exhibit 1043 - Page 22
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

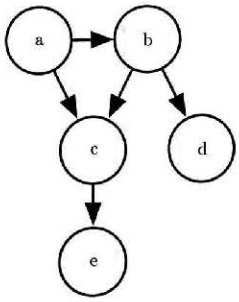


Figure 3.7: A directed graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{a}, \mathrm{b})p(\mathrm{d} \mid \mathrm{b})p(\mathrm{e} \mid \mathrm{c}). \tag{3.54}$$

This graphical model enables us to quickly see some properties of the distribution. For example, a and c interact directly, but a and e interact only indirectly via c.

See figure 3.7 for an example of a directed graph and the factorization of probability distributions it represents.

Undirected models use graphs with undirected edges, and they represent factorizations into a set of functions; unlike in the directed case, these functions are usually not probability distributions of any kind. Any set of nodes that are all connected to each other in $\mathcal{G}$ is called a clique. Each clique $\mathcal{C}^{(i)}$ in an undirected model is associated with a factor $\phi^{(i)}(\mathcal{C}^{(i)})$. These factors are just functions, not probability distributions. The output of each factor must be non-negative, but there is no constraint that the factor must sum or integrate to 1 like a probability distribution.

The probability of a configuration of random variables is **proportional** to the product of all these factors—assignments that result in larger factor values are more likely. Of course, there is no guarantee that this product will sum to 1. We therefore divide by a normalizing constant Z, defined to be the sum or integral over all states of the product of the ϕ functions, in order to obtain a normalized probability distribution:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)} \left(\mathcal{C}^{(i)} \right). \tag{3.55}$$

See figure 3.8 for an example of an undirected graph and the factorization of probability distributions it represents.

Microsoft Exhibit 1043 - Page 23
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

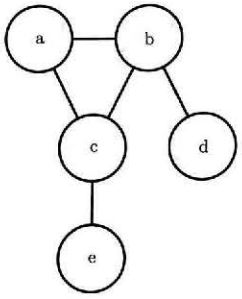


Figure 3.8: An undirected graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = \frac{1}{Z}\phi^{(1)}(\mathrm{a}, \mathrm{b}, \mathrm{c})\phi^{(2)}(\mathrm{b}, \mathrm{d})\phi^{(3)}(\mathrm{c}, \mathrm{e}). \tag{3.56}$$

This graphical model enables us to quickly see some properties of the distribution. For example, a and c interact directly, but a and e interact only indirectly via c.

Keep in mind that these graphical representations of factorizations are a language for describing probability distributions. They are not mutually exclusive families of probability distributions. Being directed or undirected is not a property of a probability distribution; it is a property of a particular **description** of a probability distribution, but any probability distribution may be described in both ways.

Throughout parts I and II of this book, we use structured probabilistic models merely as a language to describe which direct probabilistic relationships different machine learning algorithms choose to represent. No further understanding of structured probabilistic models is needed until the discussion of research topics, in part III, where we explore structured probabilistic models in much greater detail.

This chapter has reviewed the basic concepts of probability theory that are most relevant to deep learning. One more set of fundamental mathematical tools remains: numerical methods.

Microsoft Exhibit 1043 - Page 24
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 25
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

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

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 $\text{softmax}(\boldsymbol{z})$ where $\boldsymbol{z} = \boldsymbol{x} - \max_i x_i$. 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 $\log \text{softmax}(\boldsymbol{x})$ 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 softmax in a numerically stable way. The log 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

Microsoft Exhibit 1043 - Page 26
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

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

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 $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$. We usually phrase most optimization problems in terms of minimizing $f(\boldsymbol{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\boldsymbol{x})$.

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 $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$.

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

Microsoft Exhibit 1043 - Page 27
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $f(x - \epsilon \operatorname{sign}(f'(x)))$ 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, 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.

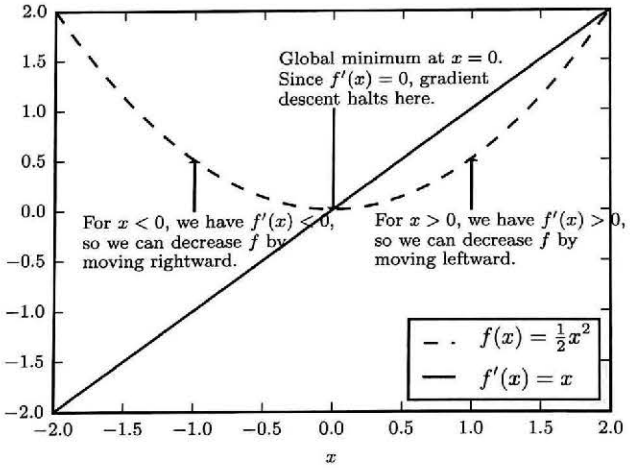


Figure 4.1: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function can be used to follow the function downhill to a minimum.

Microsoft Exhibit 1043 - Page 28
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

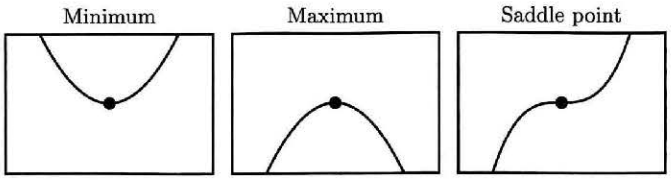


Figure 4.2: Types of critical points. Examples of the three types of critical points in one dimension. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points; a local maximum, which is higher than the neighboring points; or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

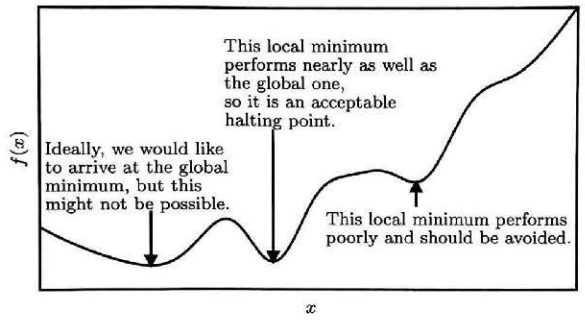


Figure 4.3: Approximate minimization. Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly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

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: $f : \mathbb{R}^n \to \mathbb{R}$. For the concept of "minimization" to make sense, there must still be only one (scalar) output.

Microsoft Exhibit 1043 - Page 29
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

For functions with multiple inputs, we must make use of the concept of **partial derivatives**. The partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ measures how f changes as only the variable x_i increases at point $\boldsymbol{x}$. 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 $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Element i of the gradient is the partial derivative of f with respect to x_i. In multiple dimensions, critical points are points where every element of the gradient is equal to zero.

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

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

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

where θ is the angle between $\boldsymbol{u}$ and the gradient. Substituting in $||\boldsymbol{u}||_2 = 1$ and ignoring factors that do not depend on $\boldsymbol{u}$, this simplifies to $\min_{\boldsymbol{u}} \cos\theta$. This is minimized when $\boldsymbol{u}$ 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

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{4.5}$$

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 $f(\boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$ 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 $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$ for $\boldsymbol{x}$.

Microsoft Exhibit 1043 - Page 30
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, then the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ of $\boldsymbol{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$.

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 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 $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that

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

Equivalently, the Hessian is the Jacobian of the gradient.

Microsoft Exhibit 1043 - Page 31
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

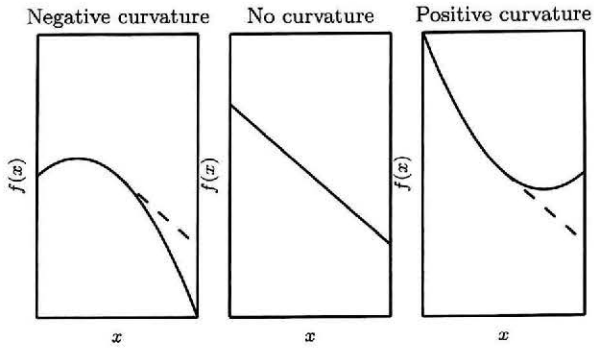


Figure 4.4: The second derivative determines the curvature of a function. Here we show quadratic functions with various curvature. The dashed line indicates the value of the cost function we would expect based on the gradient information alone as we make a gradient step downhill. With negative curvature, 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.

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

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

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 $\boldsymbol{d}$ is given by $\boldsymbol{d}^\top \boldsymbol{H} \boldsymbol{d}$. When $\boldsymbol{d}$ is an eigenvector of $\boldsymbol{H}$, the second derivative in that direction is given by the corresponding eigenvalue. For other directions of $\boldsymbol{d}$, 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 $\boldsymbol{d}$ 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

Microsoft Exhibit 1043 - Page 32
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

to the function $f(\boldsymbol{x})$ around the current point $\boldsymbol{x}^{(0)}$:

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

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

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

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 $\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$ 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 $\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$ is positive, solving for the optimal step size that decreases the Taylor series approximation of the function the most yields

$$\epsilon^* = \frac{\boldsymbol{g}^\top \boldsymbol{g}}{\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}}. \tag{4.10}$$

In the worst case, when $\boldsymbol{g}$ aligns with the eigenvector of $\boldsymbol{H}$ corresponding to the maximal eigenvalue $\lambda_{\max}$, 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

Microsoft Exhibit 1043 - Page 33
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

the second derivative test to multiple dimensions. At a critical point, where $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$, 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 $\boldsymbol{x}$ 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

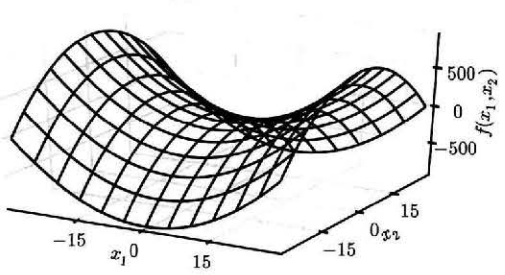


Figure 4.5: A saddle point containing both positive and negative curvature. The function in this example is $f(\boldsymbol{x}) = x_1^2 - x_2^2$. Along the axis corresponding to x_1, the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding 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.

Microsoft Exhibit 1043 - Page 34
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

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 $f(\boldsymbol{x})$ near some point $\boldsymbol{x}^{(0)}$:

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

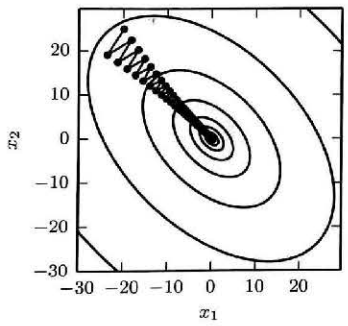


Figure 4.6: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent to minimize a quadratic function $f(\boldsymbol{x})$ whose Hessian matrix has condition number 5. This means that the direction of most curvature has five times more curvature 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.

Microsoft Exhibit 1043 - Page 35
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

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

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** $\mathcal{L}$:

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

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

Microsoft Exhibit 1043 - Page 36
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $f(\boldsymbol{x})$ over all possible values of $\boldsymbol{x}$. Instead we may wish to find the maximal or minimal value of $f(\boldsymbol{x})$ for values of $\boldsymbol{x}$ in some set $\mathbb{S}$. This is known as **constrained optimization**. Points $\boldsymbol{x}$ 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 $||\boldsymbol{x}|| \leq 1$.

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 $\boldsymbol{x}$ 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 $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^2$ with $\boldsymbol{x}$ 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.

Microsoft Exhibit 1043 - Page 37
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 38
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{x} \in \mathbb{R}^n$ 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.

Microsoft Exhibit 1043 - Page 39
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $f : \mathbb{R}^n \to \{1, \ldots, k\}$. When $y = f(\boldsymbol{x})$, the model assigns an input described by vector $\boldsymbol{x}$ 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 $\boldsymbol{x}$ 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.

Microsoft Exhibit 1043 - Page 40
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

- **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.

Microsoft Exhibit 1043 - Page 41
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{x} \in \mathbb{R}^n$, but with some entries x_i of $\boldsymbol{x}$ missing. The algorithm must provide a prediction of the values of the missing entries.

Microsoft Exhibit 1043 - Page 42
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

- **Denoising**: In this type of task, the machine learning algorithm is given in input a *corrupted example* $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$ obtained by an unknown corruption process from a *clean example* $\boldsymbol{x} \in \mathbb{R}^n$. The learner must predict the clean example $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$, or more generally predict the conditional probability distribution $p(\boldsymbol{x} \mid \tilde{\boldsymbol{x}})$.

- **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 $p_{\text{model}}(\boldsymbol{x})$ 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 $p(\boldsymbol{x})$, 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 $\boldsymbol{x}_{-i}$, are given, then we know the distribution over it is given by $p(x_i \mid \boldsymbol{x}_{-i})$. In practice, density estimation does not always enable us to solve all these related tasks, because in many cases the required operations on $p(\boldsymbol{x})$ 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

Microsoft Exhibit 1043 - Page 43
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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**.

Microsoft Exhibit 1043 - Page 44
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \ldots, \mathrm{x}_{i-1}). \tag{5.1}$$

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 $p(y \mid \mathbf{x})$ by using traditional

Microsoft Exhibit 1043 - Page 45
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

unsupervised learning technologies to learn the joint distribution $p(\mathbf{x}, y)$, then inferring

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}. \tag{5.2}$$

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 $\boldsymbol{X} \in \mathbb{R}^{150 \times 4}$, 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

Microsoft Exhibit 1043 - Page 46
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

dataset as a matrix with m rows, we describe it as a set containing m elements: $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$. This notation does not imply that any two example vectors $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(j)}$ 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 $\boldsymbol{X}$, we also provide a vector of labels $\boldsymbol{y}$, 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 $\boldsymbol{x} \in \mathbb{R}^n$ 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

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}, \tag{5.3}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ 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 $\boldsymbol{w}$ 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 $\hat{y}$.

Microsoft Exhibit 1043 - Page 47
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{x}$ by outputting $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$. 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 $\boldsymbol{X}^{(\text{test})}$ and the vector of regression targets as $\boldsymbol{y}^{(\text{test})}$.

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

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

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

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

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 $\boldsymbol{w}$ in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$. 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 $\boldsymbol{0}$:

$$\nabla_{\boldsymbol{w}} \text{MSE}_{\text{train}} = 0 \tag{5.6}$$

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

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

Microsoft Exhibit 1043 - Page 48
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

$$\Rightarrow \nabla_{\boldsymbol{w}} \left(\boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right)^\top \left(\boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right) = 0 \tag{5.9}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \left(\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - 2\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} + \boldsymbol{y}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \right) = 0 \tag{5.10}$$

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

$$\Rightarrow \boldsymbol{w} = \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \right)^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.12}$$

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

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b, \tag{5.13}$$

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 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 $\boldsymbol{x}$ with an

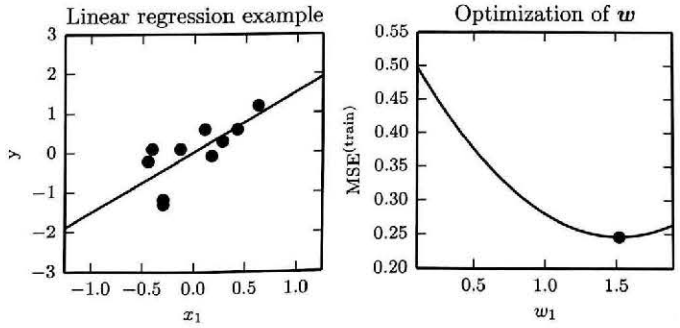


Figure 5.1: A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)*Observe that linear regression learns to set w_1 such 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.

Microsoft Exhibit 1043 - Page 49
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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,

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

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

Microsoft Exhibit 1043 - Page 50
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $p(\boldsymbol{x}, y)$ and we sample from it repeatedly to generate the training set and the test set. For some fixed value $\boldsymbol{w}$, 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.

Microsoft Exhibit 1043 - Page 51
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

$$\hat{y} = b + wx. \tag{5.15}$$

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:

$$\hat{y} = b + w_1 x + w_2 x^2. \tag{5.16}$$

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:

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

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 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

Microsoft Exhibit 1043 - Page 52
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

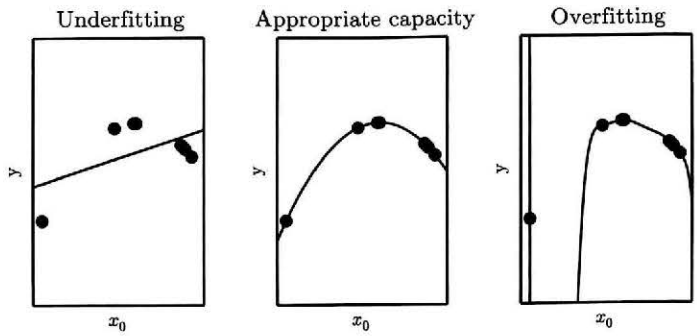


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. *(Left)*A linear function fit to the data suffers from underfitting—it cannot capture the curvature 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.

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

Microsoft Exhibit 1043 - Page 53
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{x}$ 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, 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

Microsoft Exhibit 1043 - Page 54
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

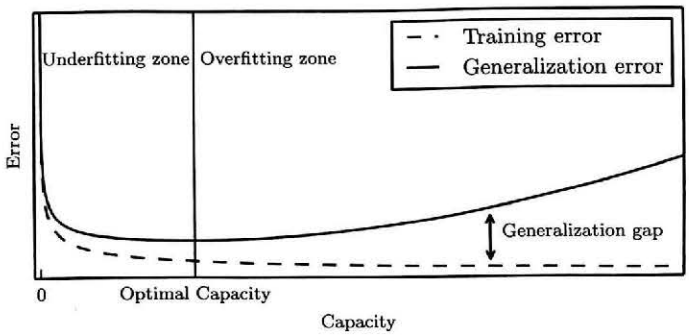


Figure 5.3: Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**. As we increase capacity, training error decreases, but the gap between training 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**.

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 $\boldsymbol{X}$ and $\boldsymbol{y}$ from the training set. When asked to classify a test point $\boldsymbol{x}$, 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 $i = \arg\min ||\boldsymbol{X}_{i,:} - \boldsymbol{x}||_2^2$. 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 $\boldsymbol{X}_{i,:}$ 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

Microsoft Exhibit 1043 - Page 55
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{x}$ to y may be inherently stochastic, or y may be a deterministic function that involves other variables besides those included in $\boldsymbol{x}$. The error incurred by an oracle making predictions from the true distribution $p(\boldsymbol{x}, y)$ 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 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.

Microsoft Exhibit 1043 - Page 56
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

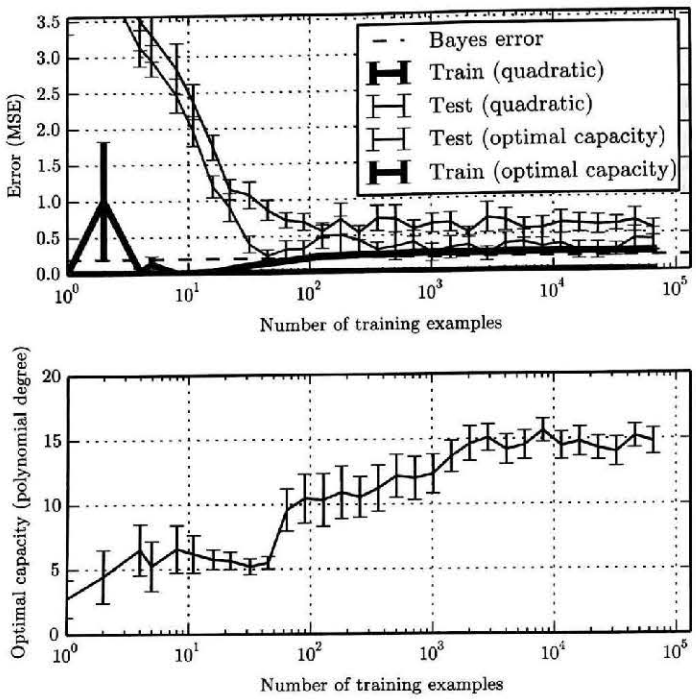


Figure 5.4: The effect of the training dataset size on the train and test error, as well as on the optimal model capacity. We constructed a synthetic regression problem based on adding a moderate amount of noise to a degree-5 polynomial, generated a single test set, and then generated several different 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.

Microsoft Exhibit 1043 - Page 57
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 58
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

comprising both the mean squared error on the training and a criterion $J(\boldsymbol{w})$ that expresses a preference for the weights to have smaller squared L^2 norm. Specifically,

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

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

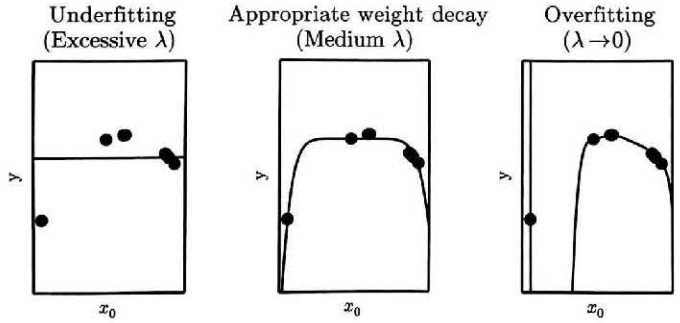


Figure 5.5: We fit a high-degree polynomial regression model to our example training set from figure 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. *(Left)*With very large λ, we can 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.

Microsoft Exhibit 1043 - Page 59
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

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

Microsoft Exhibit 1043 - Page 60
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

Microsoft Exhibit 1043 - Page 61
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

When the dataset has hundreds of thousands of examples or more, this is not a serious issue. When the dataset is too small, are 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 $\boldsymbol{\theta}$ by $\hat{\boldsymbol{\theta}}$.

Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ be a set of m independent and identically distributed (i.i.d.) data points. A **point estimator** or **statistic** is any function of the data:

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}). \tag{5.19}$$

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

Microsoft Exhibit 1043 - Page 62
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

We recognize, on the righthand side, $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$, 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 $\boldsymbol{w}$. If this prior is given by $\mathcal{N}(\boldsymbol{w}; \mathbf{0}, \frac{1}{\lambda}\boldsymbol{I}^2)$, then the log-prior term in equation 5.79 is proportional to the familiar $\lambda \boldsymbol{w}^\top \boldsymbol{w}$ weight decay penalty, plus a term that does not depend on $\boldsymbol{w}$ 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 $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. In many cases the outputs $\boldsymbol{y}$ 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 $p(y \mid \boldsymbol{x})$. We can do this simply by using maximum

Microsoft Exhibit 1043 - Page 63
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

likelihood estimation to find the best parameter vector $\boldsymbol{\theta}$ for a parametric family of distributions $p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

We have already seen that linear regression corresponds to the family

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

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:

$$p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}). \tag{5.81}$$

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 $\boldsymbol{w}^\top \boldsymbol{x} + b$. 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 $\boldsymbol{w}^\top \boldsymbol{x} + b$ is positive. Likewise, it predicts that the negative class is present when $\boldsymbol{w}^\top \boldsymbol{x} + b$ is negative.

Microsoft Exhibit 1043 - Page 64
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

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

where $\boldsymbol{x}^{(i)}$ is a training example, and $\boldsymbol{\alpha}$ is a vector of coefficients. Rewriting the learning algorithm this way enables us to replace $\boldsymbol{x}$ with the output of a given feature function $\phi(\boldsymbol{x})$ and the dot product with a function $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$ called a **kernel**. The $\cdot$ operator represents an inner product analogous to $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$. 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

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

This function is nonlinear with respect to $\boldsymbol{x}$, but the relationship between $\phi(\boldsymbol{x})$ and $f(\boldsymbol{x})$ is linear. Also, the relationship between $\boldsymbol{\alpha}$ and $f(\boldsymbol{x})$ is linear. The kernel-based function is exactly equivalent to preprocessing the data by applying $\phi(\boldsymbol{x})$ 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 $\boldsymbol{x}$ 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 $\phi(\boldsymbol{x})$ vectors and explicitly taking their dot product.

In some cases, $\phi(\boldsymbol{x})$ can even be infinite dimensional, which would result in an infinite computational cost for the naive, explicit approach. In many cases, $k(\boldsymbol{x}, \boldsymbol{x}')$ is a nonlinear, tractable function of $\boldsymbol{x}$ even when $\phi(\boldsymbol{x})$ 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.

Microsoft Exhibit 1043 - Page 65
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

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

where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the standard normal density. This kernel is also known as the **radial basis function** (RBF) kernel, because its value decreases along lines in $\boldsymbol{v}$ space radiating outward from $\boldsymbol{u}$. 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 $\boldsymbol{x}$ associated with training label y becomes a template for class y. When a test point $\boldsymbol{x}'$ is near $\boldsymbol{x}$ according to Euclidean distance, the Gaussian kernel has a large response, indicating that $\boldsymbol{x}'$ is very similar to the $\boldsymbol{x}$ 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 $\alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$ 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

Microsoft Exhibit 1043 - Page 66
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{x}$, we find the k-nearest neighbors to $\boldsymbol{x}$ in the training data $\boldsymbol{X}$. 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 $\boldsymbol{c}$ 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 $\boldsymbol{x}$ 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 $\boldsymbol{x} \in \mathbb{R}^{100}$ 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 $\boldsymbol{x}$ 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

Microsoft Exhibit 1043 - Page 67
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

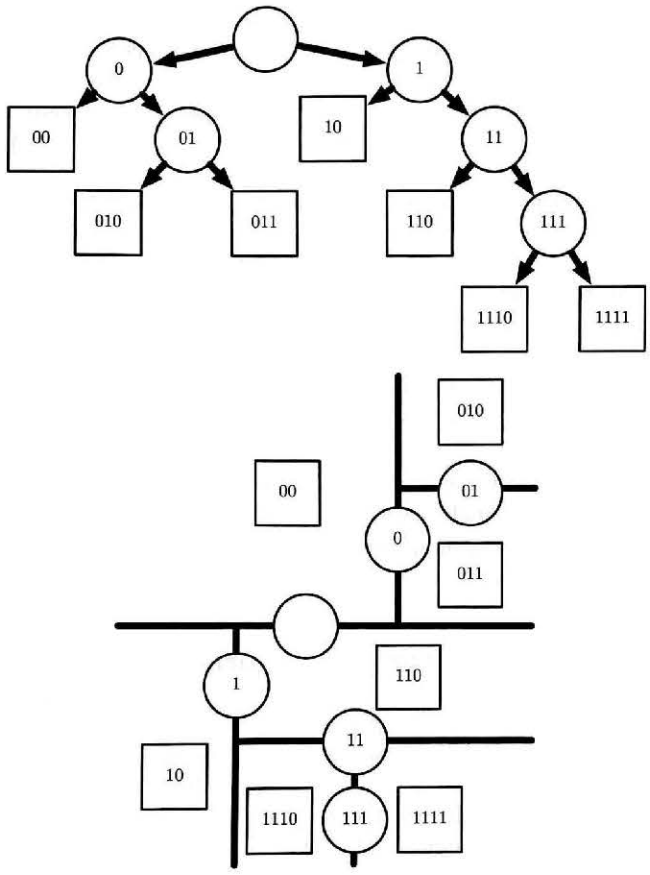


Figure 5.7: Diagrams describing how a decision tree works. *(Top)*Each node of the tree chooses to send the input example to the child node on the left (0) or to the child node on the right (1). Internal nodes are drawn as circles and leaf nodes as squares. Each node is displayed with a binary string 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.

Microsoft Exhibit 1043 - Page 68
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 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 $\boldsymbol{x}$ as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than $\boldsymbol{x}$ itself.

Microsoft Exhibit 1043 - Page 69
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

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.

Microsoft Exhibit 1043 - Page 70
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

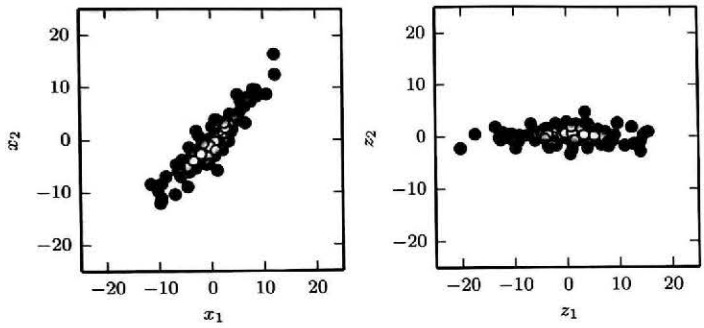


Figure 5.8: PCA learns a linear projection that aligns the direction of greatest variance with the axes of the new space. *(Left)*The original data consist of samples of $\boldsymbol{x}$. In this space, the variance might occur along directions that are not axis aligned. *(Right)*The transformed data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

PCA learns an orthogonal, linear transformation of the data that projects an input $\boldsymbol{x}$ to a representation $\boldsymbol{z}$ 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 $\boldsymbol{X}$.

Let us consider the $m \times n$ design matrix $\boldsymbol{X}$. We will assume that the data has a mean of zero, $\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$. 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 $\boldsymbol{X}$ is given by

$$\text{Var}[\boldsymbol{x}] = \frac{1}{m-1} \boldsymbol{X}^\top \boldsymbol{X}. \tag{5.85}$$

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

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

$$\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{W} \boldsymbol{\Lambda} \boldsymbol{W}^\top. \tag{5.86}$$

Microsoft Exhibit 1043 - Page 71
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{X}$. To see this, let $\boldsymbol{W}$ be the right singular vectors in the decomposition $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top$. We then recover the original eigenvector equation with $\boldsymbol{W}$ as the eigenvector basis:

$$\boldsymbol{X}^\top\boldsymbol{X} = \left(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top\right)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top = \boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top. \tag{5.87}$$

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

$$\begin{aligned}
\text{Var}[\boldsymbol{x}] &= \frac{1}{m-1}\boldsymbol{X}^\top\boldsymbol{X} && (5.88)\\
&= \frac{1}{m-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.89)\\
&= \frac{1}{m-1}\boldsymbol{W}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.90)\\
&= \frac{1}{m-1}\boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top, && (5.91)
\end{aligned}$$

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

$$\begin{aligned}
\text{Var}[\boldsymbol{z}] &= \frac{1}{m-1}\boldsymbol{Z}^\top\boldsymbol{Z} && (5.92)\\
&= \frac{1}{m-1}\boldsymbol{W}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{W} && (5.93)\\
&= \frac{1}{m-1}\boldsymbol{W}^\top\boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top\boldsymbol{W} && (5.94)\\
&= \frac{1}{m-1}\boldsymbol{\Sigma}^2, && (5.95)
\end{aligned}$$

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

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

Microsoft Exhibit 1043 - Page 72
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{z}$.

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 $\boldsymbol{h}$ representing an input $\boldsymbol{x}$. If $\boldsymbol{x}$ belongs to cluster i, then $h_i = 1$, and all other entries of the representation $\boldsymbol{h}$ 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 $\boldsymbol{x}$. 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 $\{\boldsymbol{\mu}^{(1)}, \ldots, \boldsymbol{\mu}^{(k)}\}$ 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 $\boldsymbol{x}^{(j)}$ 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

Microsoft Exhibit 1043 - Page 73
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

Microsoft Exhibit 1043 - Page 74
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

negative conditional log-likelihood of the training data can be written as

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

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

For these additive cost functions, gradient descent requires computing

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

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 $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m')}\}$ 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

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

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

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}, \tag{5.99}$$

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.

Microsoft Exhibit 1043 - Page 75
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 $G_{i,j} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$. 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 $\boldsymbol{X}$ and $\boldsymbol{y}$, the cost function

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

the model specification $p_{\text{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(y; \boldsymbol{x}^\top \boldsymbol{w} + b, 1)$, 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.

Microsoft Exhibit 1043 - Page 76
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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

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

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 $\boldsymbol{X}$ and providing an appropriate unsupervised cost and model. For example, we can obtain the first PCA vector by specifying that our loss function is

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

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

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.

Microsoft Exhibit 1043 - Page 77
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

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 $\boldsymbol{x}$ 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 $\boldsymbol{x}$, we can just return the number of training examples in the same unit volume cell as $\boldsymbol{x}$, 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

Microsoft Exhibit 1043 - Page 78
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

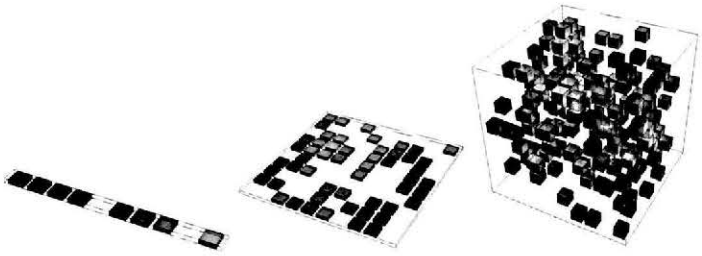


Figure 5.9: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. *(Left)*In this one-dimensional example, we have one variable for which we only care to distinguish 10 regions of interest. With enough examples 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.

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.

Microsoft Exhibit 1043 - Page 79
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

II

Deep Networks: Modern Practices

Microsoft Exhibit 1043 - Page 80
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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 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

Microsoft Exhibit 1043 - Page 81
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

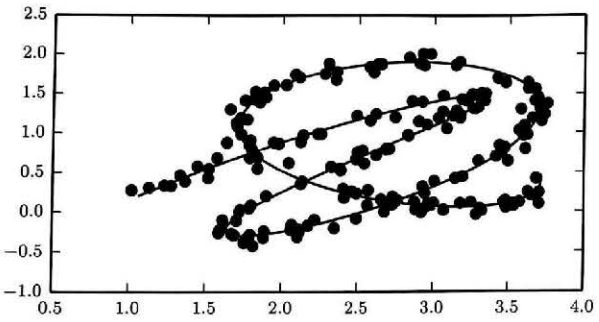


Figure 5.11: Data sampled from a distribution in a two-dimensional space that is actually concentrated near a one-dimensional manifold, like a twisted string. The solid line indicates the underlying manifold that the learner should infer.

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 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 probability 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

Microsoft Exhibit 1043 - Page 82
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

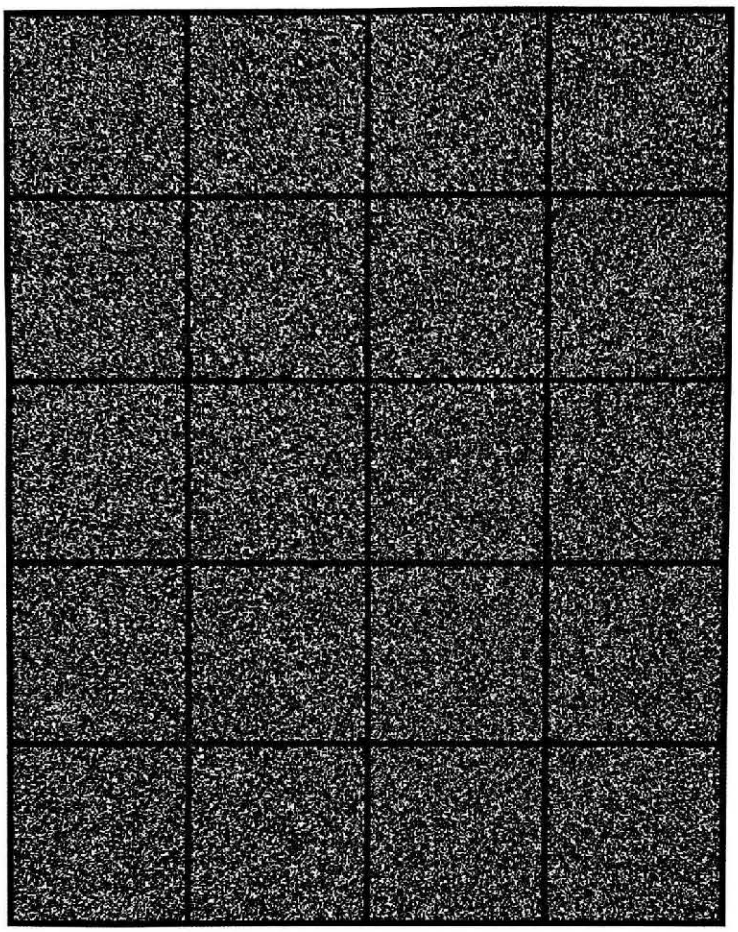


Figure 5.12: Sampling images uniformly at random (by randomly picking each pixel according to a uniform distribution) gives rise to noisy images. Although there is a nonzero probability of generating an image of a face or of any other object frequently encountered in AI applications, we never actually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

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

Microsoft Exhibit 1043 - Page 83
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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.

Microsoft Exhibit 1043 - Page 84
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

This part of the book summarizes the state of modern deep learning as it is used to solve practical applications.

Deep learning has a long history and many aspirations. Several proposed approaches have yet to entirely bear fruit. Several ambitious goals have yet to be realized. These less-developed branches of deep learning appear in the final part of the book.

This part focuses only on those approaches that are essentially working technologies that are already used heavily in industry.

Modern deep learning provides a powerful framework for supervised learning. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large models and sufficiently large datasets of labeled training examples. Other tasks, that cannot be described as associating one vector to another, or that are difficult enough that a person would require time to think and reflect in order to accomplish the task, remain beyond the scope of deep learning for now.

This part of the book describes the core parametric function approximation technology that is behind nearly all modern practical applications of deep learning. We begin by describing the feedforward deep network model that is used to represent these functions. Next, we present advanced techniques for regularization and optimization of such models. Scaling these models to large inputs such as high-resolution images or long temporal sequences requires specialization. We introduce the convolutional network for scaling to large images and the recurrent neural network for processing temporal sequences. Finally, we present general guidelines for the practical methodology involved in designing, building, and configuring an application involving deep learning and review some of its applications.

These chapters are the most important for a practitioner—someone who wants to begin implementing and using deep learning algorithms to solve real-world problems today.

Microsoft Exhibit 1043 - Page 85
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

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, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ 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 $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. 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 $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. 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

Microsoft Exhibit 1043 - Page 86
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985