O'REILLY®

Deep Learning

A PRACTITIONER'S APPROACH

FREE CHAPTERS

Josh Patterson &
Adam Gibson

O'REILLY®

Deep Learning

Although interest in machine learning has reached a high point, lofty expectations often scuttle projects before they get very far. How can machine learning—especially deep neural networks—make a real difference in your organization? This hands-on guide not only provides the most practical information available on the subject, but also helps you get started building efficient deep learning networks.

Authors Josh Patterson and Adam Gibson provide the fundamentals of deep learning—tuning, parallelization, vectorization, and building pipelines—that are valid for any library before introducing the open source Deeplearning4j (DL4J) library for developing production-class workflows. Through real-world examples, you'll learn methods and strategies for training deep network architectures and running deep learning workflows on Spark and Hadoop with DL4J.

- Dive into machine learning concepts in general, as well as deep learning in particular
- Understand how deep networks evolved from neural network fundamentals
- Explore the major deep network architectures, including Convolutional and Recurrent
- Learn how to map specific deep networks to the right problem
- Walk through the fundamentals of tuning general neural networks and specific deep network architectures
- Use vectorization techniques for different data types with DataVec, DL4J's workflow tool
- Learn how to use DL4J natively on Spark and Hadoop

"Everything a dev needs to know to get started with deep learning in the real world."

—Grant Ingersoll
CTO, Lucidworks

Josh Patterson is currently VP of Field Engineering for Skymind. Previously, Josh worked as a Principal Solutions Architect at Cloudera and as a machine learning and distributed systems engineer at the Tennessee Valley Authority.

Adam Gibson is the CTO of Skymind. Adam has worked with Fortune 500 companies, hedge funds, PR firms, and startup accelerators to create their machine learning projects. He has a strong track record helping companies handle and interpret big realtime data.

US $49.99 CAN $65.99
ISBN: 978-1-491-91425-0

Twitter: @oreillymedia
facebook.com/oreilly

ELIMINATE
DATA STORAGE BOTTLENECKS

Reduce time from data to intelligence
with a modern data platform.

Visit **purestorage.com/analytics** to learn more

PURESTORAGE®

Deep Learning

A Practitioner's Approach

This Excerpt contains Chapters 1 and 3 of the book *Deep Learning*. The full book is available on *oreilly.com* and through other retailers.

Josh Patterson and Adam Gibson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Deep Learning
by Josh Patterson and Adam Gibson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Tim McGovern
Production Editor: Nicholas Adams
Copyeditor: Bob Russell, Octal Publishing, Inc.
Proofreader: Christina Edwards
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2017: First Edition

Revision History for the First Edition
2017-07-27: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491914250* for release details.

978-1-492-02379-1

[LSI]

Table of Contents

CHAPTER 1

A Review of Machine Learning

To condense fact from the vapor of nuance
—Neal Stephenson, *Snow Crash*

The Learning Machines

Interest in machine learning has exploded over the past decade. You see machine learning in computer science programs, industry conferences, and the *Wall Street Journal* almost daily. For all the talk about machine learning, many conflate what it can do with what they wish it could do. Fundamentally, *machine learning* is using algorithms to extract information from raw data and represent it in some type of *model*. We use this model to infer things about other data we have not yet modeled.

Neural networks are one type of model for machine learning; they have been around for at least 50 years. The fundamental unit of a neural network is a *node*, which is loosely based on the biological neuron in the mammalian brain. The connections between neurons are also modeled on biological brains, as is the way these connections develop over time (with "training"). We'll dig deeper into how these models work over the next two chapters.

In the mid-1980s and early 1990s, many important architectural advancements were made in neural networks. However, the amount of time and data needed to get good results slowed adoption, and thus interest cooled. In the early 2000s computational power expanded exponentially and the industry saw a "Cambrian explosion" of computational techniques that were not possible prior to this. Deep learning emerged from that decade's explosive computational growth as a serious contender in the field, winning many important machine learning competitions. The interest has not cooled as of 2017; today, we see deep learning mentioned in every corner of machine learning.

We'll discuss our definition of deep learning in more depth in the section that follows. This book is structured such that you, the practitioner, can pick it up off the shelf and do the following:

- Review the relevant basic parts of linear algebra and machine learning
- Review the basics of neural networks
- Study the four major architectures of deep networks
- Use the examples in the book to try out variations of practical deep networks

We hope that you will find the material practical and approachable. Let's kick off the book with a quick primer on what machine learning is about and some of the core concepts you will need to better understand the rest of the book.

How Can Machines Learn?

To define how machines can learn, we need to define what we mean by "learning." In everyday parlance, when we say learning, we mean something like "gaining knowledge by studying, experience, or being taught." Sharpening our focus a bit, we can think of machine learning as using algorithms for acquiring structural descriptions from data examples. A computer learns something about the structures that represent the information in the raw data. Structural descriptions are another term for the models we build to contain the information extracted from the raw data, and we can use those structures or models to predict unknown data. Structural descriptions (or models) can take many forms, including the following:

- Decision trees
- Linear regression
- Neural network weights

Each model type has a different way of applying rules to known data to predict unknown data. Decision trees create a set of rules in the form of a tree structure and linear models create a set of parameters to represent the input data.

Neural networks have what is called a *parameter vector* representing the weights on the connections between the nodes in the network. We'll describe the details of this type of model later on in this chapter.

Machine Learning Versus Data Mining

Data mining has been around for many decades, and like many terms in machine learning, it is misunderstood or used poorly. For the context of this book, we consider the practice of "data mining" to be "extracting information from data." Machine learning differs in that it refers to the algorithms used during data mining for acquiring the structural descriptions from the raw data. Here's a simple way to think of data mining:

- To learn concepts
  - we need examples of raw data
- Examples are made of rows or instances of the data
  - Which show specific patterns in the data
- The machine learns concepts from these patterns in the data
  - Through algorithms in machine learning

Overall, this process can be considered "data mining."

Arthur Samuel, a pioneer in artificial intelligence (AI) at IBM and Stanford, defined machine learning as follows:

> [The f]ield of study that gives computers the ability to learn without being explicitly programmed.

Samuel created software that could play checkers and adapt its strategy as it learned to associate the probability of winning and losing with certain dispositions of the board. That fundamental schema of searching for patterns that lead to victory or defeat and then recognizing and reinforcing successful patterns underpins machine learning and AI to this day.

The concept of machines that can learn to achieve goals on their own has captivated us for decades. This was perhaps best expressed by the modern grandfathers of AI, Stuart Russell and Peter Norvig, in their book *Artificial Intelligence: A Modern Approach*:

> How is it possible for a slow, tiny brain, whether biological or electronic, to perceive, understand, predict, and manipulate a world far larger and more complicated than itself?

This quote alludes to ideas around how the concepts of learning were inspired from processes and algorithms discovered in nature. To set deep learning in context visually, Figure 1-1 illustrates our conception of the relationship between AI, machine learning, and deep learning.

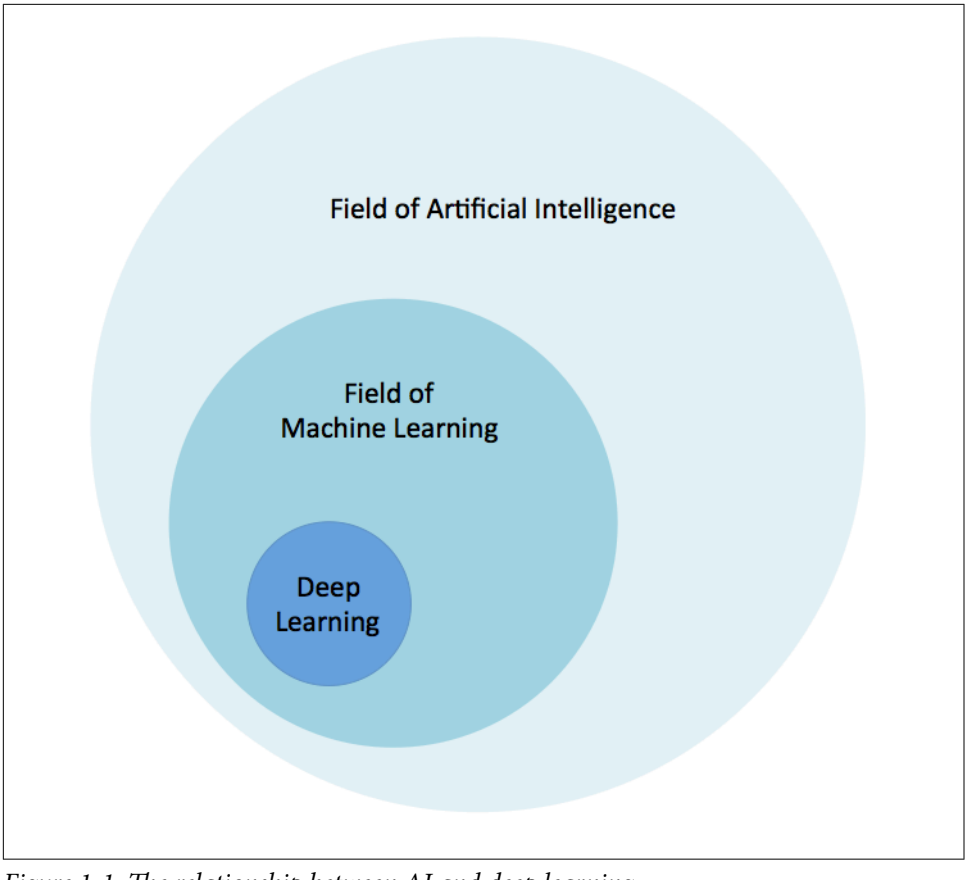

Figure 1-1. The relationship between AI and deep learning

The field of AI is broad and has been around for a long time. Deep learning is a subset of the field of machine learning, which is a subfield of AI. Let's now take a quick look at another of the roots of deep learning: how neural networks are inspired by biology.

Biological Inspiration

Biological neural networks (brains) are composed of roughly 86 billion neurons connected to many other neurons.

Total Connections in the Human Brain

Researchers conservatively estimate there are more than 500 trillion connections between neurons in the human brain. Even the largest artificial neural networks today don't even come close to approaching this number.

From an information processing point of view a biological neuron is an excitable unit that can process and transmit information via electrical and chemical signals. A neuron in the biological brain is considered a main component of the brain, spinal cord of the central nervous system, and the ganglia of the peripheral nervous system. As we'll see later in this chapter, artificial neural networks are far simpler in their comparative structure.

Comparing Biological with Artificial

Biological neural networks are considerably more complex (several orders of magnitude) than the artificial neural network versions!

There are two main properties of artificial neural networks that follow the general idea of how the brain works. First is that the most basic unit of the neural network is the *artificial neuron* (or *node* in shorthand). Artificial neurons are modeled on the biological neurons of the brain, and like biological neurons, they are stimulated by inputs. These artificial neurons pass on some—but not all—information they receive to other artificial neurons, often with transformations. As we progress through this chapter, we'll go into detail about what these transformations are in the context of neural networks.

Second, much as the neurons in the brain can be trained to pass forward only signals that are useful in achieving the larger goals of the brain, we can train the neurons of a neural network to pass along only useful signals. As we move through this chapter we'll build on these ideas and see how artificial neural networks are able to model their biological counterparts through bits and functions.

Biological Inspiration Across Computer Science

Biological inspiration is not limited to artificial neural networks in computer science. Over the past 50 years, academic research has explored other topics in nature for computational inspiration, such as the following:

- Ants
- Termites[1]
- Bees
- Genetic algorithms

Ant colonies, for instance, have been by researchers to be a powerful decentralized computer in which no single ant is a central point of failure. Ants constantly switch

1 Patterson. 2008. "TinyTermite: A Secure Routing Algorithm" and Sartipi and Patterson. 2009. "TinyTermite: A Secure Routing Algorithm on Intel Mote 2 Sensor Network Platform."

tasks to find near optimal solutions for load balancing through meta-heuristics such as quantitative stigmergy. Ant colonies are able to perform midden tasks, defense, nest construction, and forage for food while maintaining a near-optimal number of workers on each task based on the relative need with no individual ant directly coordinating the work.

What Is Deep Learning?

Deep learning has been a challenge to define for many because it has changed forms slowly over the past decade. One useful definition specifies that deep learning deals with a "neural network with more than two layers." The problematic aspect to this definition is that it makes deep learning sound as if it has been around since the 1980s. We feel that neural networks had to transcend architecturally from the earlier network styles (in conjunction with a lot more processing power) before showing the spectacular results seen in more recent years. Following are some of the facets in this evolution of neural networks:

- More neurons than previous networks
- More complex ways of connecting layers/neurons in NNs
- Explosion in the amount of computing power available to train
- Automatic feature extraction

For the purposes of this book, we'll define deep learning as neural networks with a large number of parameters and layers in one of four fundamental network architectures:

- Unsupervised pretrained networks
- Convolutional neural networks
- Recurrent neural networks
- Recursive neural networks

There are some variations of the aforementioned architectures—a hybrid convolutional and recurrent neural network, for example–as well. For the purpose of this book, we'll consider the four listed architectures as our focus.

Automatic feature extraction is another one of the great advantages that deep learning has over traditional machine learning algorithms. By feature extraction, we mean that the network's process of deciding which characteristics of a dataset can be used as indicators to label that data reliably. Historically, machine learning practitioners have spent months, years, and sometimes decades of their lives manually creating exhaustive feature sets for the classification of data. At the time of deep learning's Big Bang beginning in 2006, state-of-the-art machine learning algorithms had absorbed decades of human effort as they accumulated relevant features by which to classify input. Deep learning has surpassed those conventional algorithms in accuracy for almost

every data type with minimal tuning and human effort. These deep networks can help data science teams save their blood, sweat, and tears for more meaningful tasks.

Going Down the Rabbit Hole

Deep learning has penetrated the computer science consciousness beyond most techniques in recent history. This is in part due to how it has shown not only top-flight accuracy in machine learning modeling, but also demonstrated generative mechanics that fascinate even the noncomputer scientist. One example of this would be the art generation demonstrations for which a deep network was trained on a particular famous painter's works, and the network was able to render other photographs in the painter's unique style, as demonstrated in Figure 1-2.

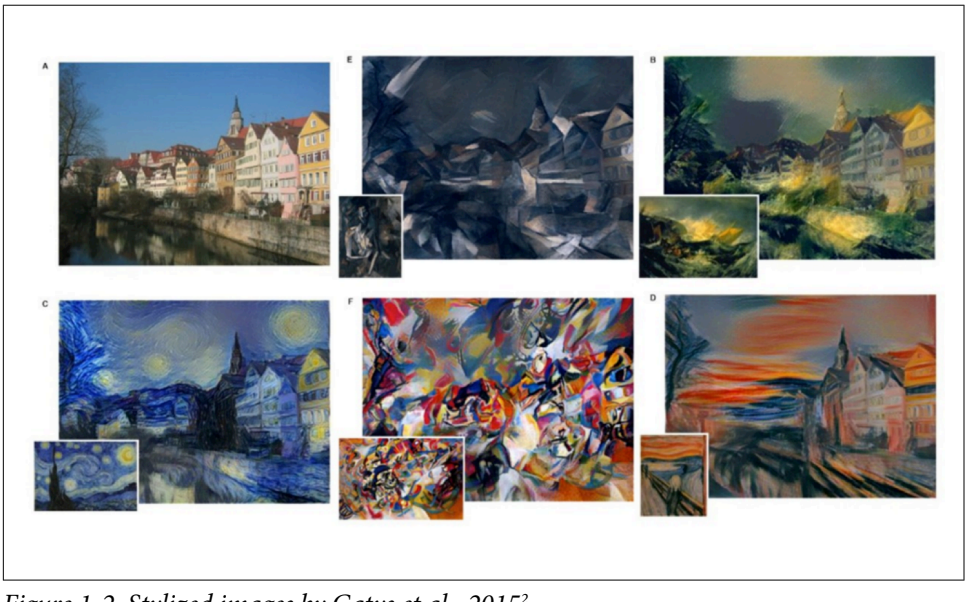

Figure 1-2. Stylized images by Gatys et al., 2015[2]

This begins to enter into many philosophical discussions, such as, "can machines be creative?" and then "what is creativity?" We'll leave those questions for you to ponder at a later time. Machine learning has evolved over the years, like the seasons change: subtle but steady until you wake up one day and a machine has become a champion on *Jeopardy* or beat a Go Grand Master.

Can machines be intelligent and take on human-level intelligence? What is AI and how powerful could it become? These questions have yet to be answered and will not

2 Gatys et. al, 2015. "A Neural Algorithm of Artistic Style."

be completely answered in this book. We simply seek to illustrate some of the shards of machine intelligence with which we can imbue our environment today through the practice of deep learning.

For an Extended Discussion on AI

If you would like to read more about AI, take a look at Appendix A.

Framing the Questions

The basics of applying machine learning are best understood by asking the correct questions to begin with. Here's what we need to define:

- What is the input data from which we want to extract information (model)?
- What kind of model is most appropriate for this data?
- What kind of answer would we like to elicit from new data based on this model?

If we can answer these three questions, we can set up a machine learning workflow that will build our model and produce our desired answers. To better support this workflow, let's review some of the core concepts we need to be aware of to practice machine learning. Later, we'll come back to how these come together in machine learning and then use that information to better inform our understanding of both neural networks and deep learning.

The Math Behind Machine Learning: Linear Algebra

Linear algebra is the bedrock of machine learning and deep learning. Linear algebra provides us with the mathematical underpinnings to solve the equations we use to build models.

A great primer on linear algebra is James E. Gentle's *Matrix Algebra: Theory, Computations, and Applications in Statistics*.

Let's take a look at some core concepts from this field before we move on starting with the basic concept called a *scalar*.

Scalars

In mathematics, when the term scalar is mentioned, we are concerned with elements in a vector. A scalar is a real number and an element of a field used to define a vector space.

In computing, the term scalar is synonymous with the term variable and is a storage location paired with a symbolic name. This storage location holds an unknown quantity of information called a *value*.

Vectors

For our use, we define a vector as follows:

> *For a positive integer* n*, a vector is an* n*-tuple, ordered (multi)set or array of* n *numbers, called elements or scalars.*

What we're saying is that we want to create a data structure called a vector via a process called *vectorization*. The number of elements in the vector is called the "order" (or "length") of the vector. Vectors also can represent points in n-dimensional space. In the spatial sense, the Euclidean distance from the origin to the point represented by the vector gives us the "length" of the vector.

In mathematical texts, we often see vectors written as follows:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix}$$

Or:

$$x = \begin{bmatrix} x_1, x_2, x_3, \dots, x_n \end{bmatrix}$$

There are many different ways to handle the vectorization, and you can apply many preprocessing steps, giving us different grades of effectiveness on the output models. We cover more on the topic of converting raw data into vectors later in this chapter and then more fully in Chapter 5.

Matrices

Consider a matrix to be a group of vectors that all have the same dimension (number of columns). In this way a matrix is a two-dimensional array for which we have rows and columns.

If our matrix is said to be an $n \times m$ matrix, it has n rows and m columns. Figure 1-3 shows a 3×3 matrix illustrating the dimensions of a matrix. Matrices are a core structure in linear algebra and machine learning, as we'll show as we progress through this chapter.

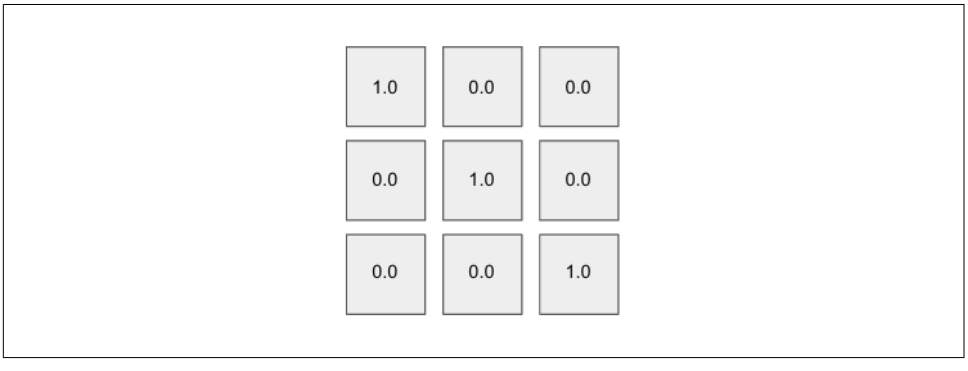

Figure 1-3. A 3 x 3 matrix

Tensors

A *tensor* is a multidimensional array at the most fundamental level. It is a more general mathematical structure than a vector. We can look at a vector as simply a subclass of tensors.

With tensors, the rows extend along the y-axis and the columns along the x-axis. Each axis is a dimension, and tensors have additional dimensions. Tensors also have a rank. Comparatively, a scalar is of rank 0 and a vector is rank 1. We also see that a matrix is rank 2. Any entity of rank 3 and above is considered a tensor.

Hyperplanes

Another linear algebra object you should be aware of is the *hyperplane*. In the field of geometry, the hyperplane is a subspace of one dimension less than its ambient space. In a three-dimensional space, the hyperplanes would have two dimensions. In two-dimensional space we consider a one-dimensional line to be a hyperplane.

A hyperplane is a mathematical construct that divides an n-dimensional space into separate "parts" and therefore is useful in applications like classification. Optimizing the parameters of the hyperplane is a core concept in linear modeling, as you'll see further on in this chapter.

Relevant Mathematical Operations

In this section, we briefly review common linear algebra operations you should know.

Dot product

A core linear algebra operation we see often in machine learning is the *dot product*. The dot product is sometimes called the "scalar product" or "inner product." The dot product takes two vectors of the same length and returns a single number. This is done by matching up the entries in the two vectors, multiplying them, and then summing up the products thus obtained. Without getting too mathematical (immediately), it is important to mention that this single number encodes a lot of information.

To begin with, the dot product is a measure of how big the individual elements are in each vector. Two vectors with rather large values can give rather large results, and two vectors with rather small values can give rather small values. When the relative values of these vectors are accounted for mathematically with something called *normalization*, the dot product is a measure of how similar these vectors are. This mathematical notion of a dot product of two normalized vectors is called the *cosine similarity*.

Element-wise product

Another common linear algebra operation we see in practice is the *element-wise product* (or the "Hadamard product"). This operation takes two vectors of the same length and produces a vector of the same length with each corresponding element multiplied together from the two source vectors.

Outer product

This is known as the "tensor product" of two input vectors. We take each element of a column vector and multiply it by all of the elements in a row vector creating a new row in the resultant matrix.

Converting Data Into Vectors

In the course of working in machine learning and data science we need to analyze all types of data. A key requirement is being able to take each data type and represent it as a vector. In machine learning we use many types of data (e.g., text, time-series, audio, images, and video).

So, why can't we just feed raw data to our learning algorithm and let it handle everything? The issue is that machine learning is based on linear algebra and solving sets of equations. These equations expect floating-point numbers as input so we need a way to translate the raw data into sets of floating-point numbers. We'll connect these concepts together in the next section on solving these sets of equations. An example of raw data would be the canonical iris dataset:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

Another example might be a raw text document:

```
Go, Dogs. Go!
Go on skates
or go by bike.
```

Both cases involve raw data of different types, yet both need some level of vectorization to be of the form we need to do machine learning. At some point, we want our input data to be in the form of a matrix but we can convert the data to intermediate representations (e.g., "svmlight" file format, shown in the code example that follows). We want our machine learning algorithm's input data to look more like the serialized sparse vector format svmlight, as shown in the following example:

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468 4:0.
  5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0
  .7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0
  .5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0
  .5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149 4:0
  .5217391304347826
0.0 2:0.7500000000000002
```

This format can quickly be read into a matrix and a column vector for the labels (the first number in each row in the preceding example). The rest of the indexed numbers in the row are inserted into the proper slot in the matrix as "features" at runtime to get ready for various linear algebra operations during the machine learning process. We'll discuss the process of vectorization in more detail in Chapter 8.

Here's a very common question: "why do machine learning algorithms want the data represented (typically) as a (sparse) matrix?" To understand that, let's make a quick detour into the basics of solving systems of equations.

Solving Systems of Equations

In the world of linear algebra, we are interested in solving systems of linear equations of the form:

$$Ax = b$$

where A is a matrix of our set of input row vectors and b is the column vector of labels for each vector in the A matrix. If we take the first three rows of serialized sparse output from the previous example and place the values in its linear algebra form, it looks like this:

Column 1	Column 2	Column 3	Column 4
0.7500000000000001	0.41666666666666663	0.702127659574468	0.5652173913043479
0.6666666666666666	0.5	0.9148936170212765	0.6956521739130436
0.45833333333333326	0.3333333333333336	0.8085106382978723	0.7391304347826088

This matrix of numbers is our A variable in our equation, and each independent value or value in each row is considered a feature of our input data.

What Is a Feature?

A feature in machine learning is any column value in the input matrix A that we're using as an independent variable. Features can be taken straight from the source data, but most of the time we're going to use some sort of transformation to get the raw input data into a form that is more appropriate for modeling.

An example would be a column of input that has four different text labels in the source data. We'd need to scan all of the input data and index the labels being used. We'd then need to normalize these values (0, 1, 2, 3) between 0.0 and 1.0 based on each label's index for every row's column value. These types of transforms greatly help machine learning find better solutions to modeling problems. We'll see more techniques for vectorization transforms in Chapter 5.

We want to find coefficients for each column in a given row for a predictor function that give us the output b, or the label for each row. The labels from the serialized sparse vectors we looked at earlier would be as follows:

Labels
1.0
2.0
2.0

The coefficients mentioned earlier become the x column vector (also called the *parameter vector*) shown in Figure 1-4.

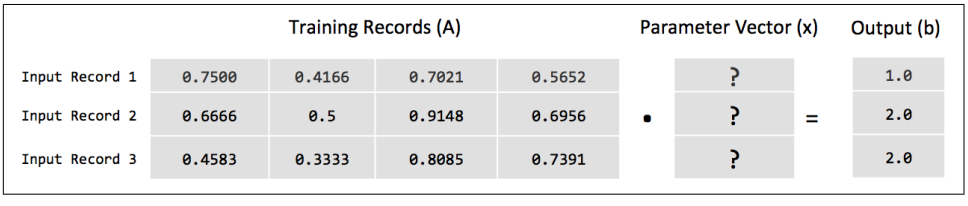

Figure 1-4. Visualizing the equation $Ax = b$

This system is said to be "consistent" if there exists a parameter vector x such that the solution to this equation can be directly written as follows:

$$x = A^{-1}b$$

It's important to delineate the expression $x = A^{-1}b$ from the method of actually computing the solution. This expression only represents the solution itself. The variable A^{-1} is the matrix A inverted and is computed through a process called *matrix inversion*. Given that not all matrices can be inverted, we'd like a method to solve this equation that does not involve matrix inversion. One method is called *matrix decomposition*. An example of matrix decomposition in solving systems of linear equations is using lower upper (LU) decomposition to solve for the matrix A. Beyond matrix decomposition, let's take a look at the general methods for solving sets of linear equations.

Methods for solving systems of linear equations

There are two general methods for solving a system of linear equations. The first is called the "direct method," in which we know algorithmically that there are a fixed number of computations. The other approach is a class of methods known as *iterative methods*, in which through a series of approximations and a set of termination conditions we can derive the parameter vector x. The direct class of methods is particularly effective when we can fit all of the training data (A and b) in memory on a single computer. Well-known examples of the direct method of solving sets of linear equations are *Gaussian Elimination* and the *Normal Equations*.

Iterative methods

The iterative class of methods is particularly effective when our data doesn't fit into the main memory on a single computer, and looping through individual records from disk allows us to model a much larger amount of data. The canonical example of iterative methods most commonly seen in machine learning today is *Stochastic Gradient Descent* (SDG), which we discuss later in this chapter. Other techniques in this space are *Conjugate Gradient Methods* and *Alternating Least Squares* (discussed further in Chapter 2). Iterative methods also have been shown to be effective in scale-out methods, for which we not only loop through local records, but the entire dataset is sharded across a cluster of machines, and periodically the parameter vector is averaged across all agents and then updated at each local modeling agent (described in more detail in Chapter 9).

Iterative methods and linear algebra

At the mathematical level, we want to be able to operate on our input dataset with these algorithms. This constraint requires us to convert our raw input data into the input matrix *A*. This quick overview of linear algebra gives us the "why" for going through the trouble to vectorize data. Throughout this book, we show code examples of converting the raw input data into the input matrix *A*, giving you the "how." The mechanics of how we vectorize our data also affects the results of the learning process. As we'll see later in the book, how we handle data in the preprocess stage before vectorization can create more accurate models.

The Math Behind Machine Learning: Statistics

Let's review just enough statistics to let this chapter move forward. We need to highlight some basic concepts in statistics, such as the following:

- Probabilities
- Distributions
- Likelihood

There are also some other basic relationships we'd like to highlight in descriptive statistics and inferential statistics. Descriptive statistics include the following:

- Histograms
- Boxplots
- Scatterplots
- Mean
- Standard deviation
- Correlation coefficient

This contrasts with how inferential statistics are concerned with techniques for generalizing from a sample to a population. Here are some examples of inferential statistics:

- p-values
- credibility intervals

The relationship between probability and inferential statistics:

- Probability reasons from the population to the sample (deductive reasoning)
- Inferential statistics reason from the sample to the population

Before we can understand what a specific sample tells us about the source population, we need to understand the uncertainty associated with taking a sample from a given population.

Regarding general statistics, we won't linger on what is an inherently broad topic already covered in depth by other books. This section is in no way meant to serve as a true statistics review; rather, it is designed to direct you toward relevant topics that you can investigate in greater depth from other resources. With that disclaimer out of the way, let's begin by defining probability in statistics.

Probability

We define probability of an event *E* as a number always between 0 and 1. In this context, the value 0 infers that the event *E* has no chance of occurring, and the value 1 means that the event *E* is certain to occur. Many times we'll see this probability expressed as a floating-point number, but we also can express it as a percentage between 0 and 100 percent; we will not see valid probabilities lower than 0 percent and greater than 100 percent. An example would be a probability of 0.35 expressed as 35 percent (e.g., 0.35 x 100 == 35 percent).

The canonical example of measuring probability is observing how many times a fair coin flipped comes up heads or tails (e.g., 0.5 for each side). The probability of the sample space is always 1 because the sample space represents all possible outcomes for a given trial. As we can see with the two outcomes ("heads" and its complement, "tails") for the flipped coin, 0.5 + 0.5 == 1.0 because the total probability of the sample space must always add up to 1. We express the probability of an event as follows:

$$P(E) = 0.5$$

And we read this like so:

> The probability of an event *E* is 0.5

Probability Versus Odds, Explained

Many times practitioners new to statistics or machine learning will conflate the meaning of probability and odds. Before we proceed, let's clarify this here.

The probability of an event *E* is defined as:

$$P(E) = (\text{Chances for } E) / (\text{Total Chances})$$

We see this in the example of drawing an ace card (4) out of a deck of cards (52) where we'd have this:

$$4/52 = 0.077$$

Conversely, odds are defined as:

$$(\text{Chances for } E) : (\text{Chances Against } E)$$

Now our card example becomes the "odds of drawing an ace":

$$4 : (52 - 4) = 1/12 = 0.0833333...$$

The primary difference here is the choice of denominator (Total Chances versus Chances Against) making these two distinct concepts in statistics.

Probability is at the center of neural networks and deep learning because of its role in feature extraction and classification, two of the main functions of deep neural networks. For a larger review of statistics, check out O'Reilly's *Statistics in a Nutshell: A Desktop Quick Reference* by Boslaugh and Watters.

Further Defining Probability: Bayesian Versus Frequentist

There are two different approaches in statistics called *Bayesianism* and *frequentism*. The basic difference between the approaches is how probability is defined.

With frequentists, probability only has meaning in the context of repeating a measurement. As we measure something, we'll see slight variations due to variances in the equipment we use to collect data. As we measure something a large number of times, the frequency of the given value indicates the probability of measuring that value.

With the Bayesian approach, we extend the idea of probability to cover aspects of certainty about statements. The probability gives us a statement of our knowledge of what the measurement result will be. For Bayesians, our own knowledge about an event is fundamentally related to probability.

Frequentists rely on many, many blind trials of an experiment before making statements about an estimate for a variable. Bayesians, on the other hand, deal in "beliefs" (in mathematical terms, "distributions") about the variable and update their beliefs about the variable as new information comes in.

Conditional Probabilities

When we want to know the probability of a given event based on the existing presence of another event occurring, we express this as a *conditional probability*. This is expressed in literature in the form:

$$P(E \mid F)$$

where:

E is the event for which we're interested in a probability.

F is the event that has already occurred.

An example would be expressing how a person with a healthy heart rate has a lower probability of ICU death during a hospital visit:

P(ICU Death | Poor Heart Rate) > *P*(ICU Death | Healthy Heart Rate)

Sometimes, we'll hear the second event, *F*, referred to as the "condition." Conditional probability is interesting in machine learning and deep learning because we're often interested in when multiple things are happening and how they interact. We're interested in conditional probabilities in machine learning in the context in which we'd learn a classifier by learning

$$P (E \mid F)$$

where *E* is our label and *F* is a number of attributes about the entity for which we're predicting *E*. An example would be predicting mortality (here, *E*) given that measurements taken in the ICU for each patient (here, *F*).

Bayes's Theorem

One of the more common applications of conditional probabilities is Bayes's theorem (or Bayes's formula). In the field of medicine, we see it used to calculate the probability that a patient who tests positive on a test for a specific disease actually has the disease.

We define Bayes's formula for any two events, A and B, as:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

Posterior Probability

In Bayesian statistics, we call the posterior probability of the random event the conditional probability we assign after the evidence is considered. Posterior probability distribution is defined as the probability distribution of an unknown quantity conditional on the evidence collected from an experiment treated as a random variable. We see this concept in action with softmax activation functions (explained later in this chapter), in which raw input values are converted to posterior probabilities.

Distributions

A probability distribution is a specification of the stochastic structure of random variables. In statistics, we rely on making assumptions about how the data is distributed to make inferences about the data. We want a formula that specifies how frequent values of observations in the distribution are and how values can be taken by points in the distribution. A common distribution is known as the *normal distribution* (also called *Gaussian distribution*, or the *bell curve*). We like to fit a dataset to a distribution because if the dataset is reasonably close to the distribution, we can make assumptions based on the theoretical distribution in how we operate with the data.

We classify distributions as *continuous* or *discrete*. A discrete distribution has data that can assume only certain values. In a continuous distribution, data can be any value within the range. An example of a continuous distribution would be normal distribution. An example of a discrete distribution would be binomial distribution.

Normal distribution allows us to assume sampling distributions of statistics (e.g., "sample mean") are normally distributed under specified conditions. The normal distribution (see Figure 1-5), or *Gaussian distribution*, was named after the eighteenth-century mathematician and physicist Karl Gauss. Normal distribution is defined by its mean and standard deviation and has generally the same shape across all variations.

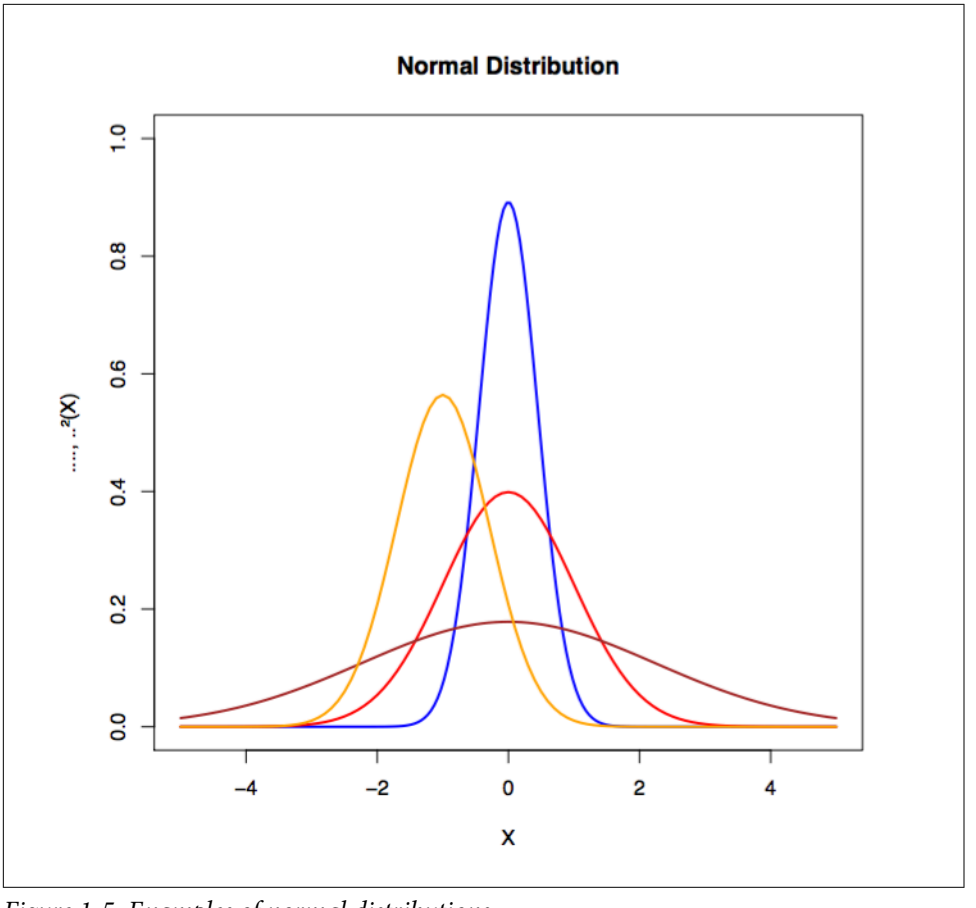

Figure 1-5. Examples of normal distributions

Other relevant distributions in machine learning include the following:

- Binomial distribution
- Inverse Gaussian distribution
- Log normal distribution

Distribution of the training data in machine learning is important to understand how to vectorize the data for modeling.

Central Limit Theorem

If the sample size is sufficiently large, the sampling distribution of the sample mean approximates the normal distribution. This holds true despite the distribution of the population from which the samples were collected.

Based on this fact, we can make statistical inferences using tests based on the approximate normality of the mean. We see this hold true regardless of whether the sample is drawn from a population that is not normally distributed.

In computer science, we see this used where an algorithm repeatedly draws samples of specified size from a nonnormal population. When we graph the histogram of the sample population of the draws from a normal distribution, we can see this effect in action.

A long-tailed distribution (such as Zipf, power laws, and Pareto distributions) is a scenario in which a high-frequency population is followed by a low-frequency population that gradually decreases in asymptotic fashion. These distributions were discovered by Benoit Mandelbrot in the 1950s and later popularized by the writer Chris Anderson in his book *The Long Tail: Why the Future of Business is Selling Less of More*.

An example would be ranking the items a retailer sells among which a few items are exceptionally popular and then we see a large number of unique items with relatively small quantities sold. This rank-frequency distribution (primarily of popularity or "how many were sold") often forms power laws. From this perspective, we can consider them to be long-tailed distributions.

We see these long-tailed distributions manifested in the following:

Earthquake damage
: The damage becomes worse as the scale of the quake increases, so worse case shifts.

Crop yields
: We sometimes see events outside of the historical record, whereas our model tends to be tuned around the mean.

Predicting fatality post ICU visit
: We can have events far outside the scope of what happens inside the ICU visit that affect mortality.

These examples are relevant in the context of this book for classification problems because most statistical models depend on inference from lots of data. If the more interesting events occur out in the tail of the distribution and we don't have this represented in the training sample data, our model might perform unpredictably. This effect can be enhanced in nonlinear models such as neural networks. We'd consider

this situation the special case of the "in sample/out of sample" problem. Even a seasoned machine learning practitioner can be surprised at how well a model performs on a skewed training data sample yet fails to generalize well on the larger population of data.

Long-tailed distributions deal with the real possibility of events occurring that are five times the standard deviation. We must be mindful to get a decent representation of events in our training data to prevent overfitting the training data. We'll look in greater detail at ways to do this later when we talk about overfitting and then in Chapter 4 on tuning.

Samples Versus Population

A population of data is defined as all of the units we'd like to study or model in our experiment. An example would be defining our population of study as "all Java programmers in the state of Tennessee."

A sample of data is a subset of the population of data that hopefully represents the accurate distribution of the data without introducing sampling bias (e.g., skewing the sample distribution based on how we sampled the population).

Resampling Methods

Bootstrapping and *cross-validation* are two common methods of resampling in statistics that are useful to machine learning practitioners. In the context of machine learning with bootstrapping, we're drawing random samples from another sample to generate a new sample that has a balance between the number of samples per class. This is useful when we'd like to model against a dataset with highly unbalanced classes.

Cross-validation (also called *rotation estimation*) is a method to estimate how well a model generalizes on a training dataset. In cross-validation we split the training dataset into *N* number of splits and then separate the splits into training and test groups. We train on the training group of splits and then test the model on the test group of splits. We rotate the splits between the two groups many times until we've exhausted all the variations. There is no hard number for the number of splits to use but researchers have found 10 splits to work well in practice. It also is common to see a separate portion of the held-out data used as a validation dataset during training.

Selection Bias

In *selection bias* we're dealing with a sampling method that does not have proper randomization and skews the sample in a way such that the sample is not representative of the population we'd like to model. We need to be aware of selection bias when

resampling datasets so that we don't introduce bias into our models that will lower our model's accuracy on data from the larger population.

Likelihood

When we discuss the likeliness that an event will occur yet do not specifically reference its numeric probability, we're using the informal term, *likelihood*. Typically, when we use this term, we're talking about an event that has a reasonable probability of happening but still might not. There also might be factors not yet observed that will influence the event, as well. Informally, likelihood is also used as a synonym for probability.

How Does Machine Learning Work?

In a previous section on solving systems of linear equations, we introduced the basics of solving $Ax = b$. Fundamentally, machine learning is based on algorithmic techniques to minimize the error in this equation through *optimization*.

In optimization, we are focused on changing the numbers in the x column vector (parameter vector) until we find a good set of values that gives us the closest outcomes to the actual values. Each weight in the weight matrix will be adjusted after the loss function calculates the error (based on the actual outcome, as shown earlier, as the b column vector) produced by the network. An error matrix attributing some portion of the loss to each weight will be multiplied by the weights themselves.

We discuss SDG further on in this chapter as one of the major methods to perform machine learning optimization, and then we'll connect these concepts to other optimization algorithms as the book progresses. We'll also cover the basics of hyperparameters such as regularization and learning rate.

Regression

Regression refers to functions that attempt to predict a real value output. This type of function estimates the dependent variable by knowing the independent variable. The most common class of regression is *linear regression*, based on the concepts we've previously described in modeling systems of linear equations. Linear regression attempts to come up with a function that describes the relationship between x and y, and, for known values of x, predicts values of y that turn out to be accurate.

Setting up the model

The prediction of a linear regression model is the linear combination of coefficients (from the parameter vector x) and then input variables (features from the input vector). We can model this by using the following equation:

$$y = a + Bx$$

where a is the y-intercept, B is the input features, and x is the parameter vector.

This equation expands to the following:

$$y = a + b_0 * x_0 + b_1 * x_1 + \ldots + b_n * x_n$$

A simple example of a problem that linear regression solves would be predicting how much we'd spend per month on gasoline based on the length of our commute. Here, what you pay at the tank is a function of how far you drive. The gas cost is the dependent variable and the length of the commute is the independent variable. It's reasonable to keep track of these two quantities and then define a function, like so:

$$cost = f\,(distance)$$

This allows us to reasonably predict our gasoline spending based on mileage. In this example, we'd consider distance to be our independent variable and cost to be the dependent variable in our model f.

Here are some other examples of linear regression modeling:

- Predicting weight as a function of height
- Predicting a house's sale price based on its square footage

Visualizing linear regression

Visually, we can represent linear regression as finding a line that comes as close to as many points as possible in a scatterplot of data, as demonstrated in Figure 1-6.

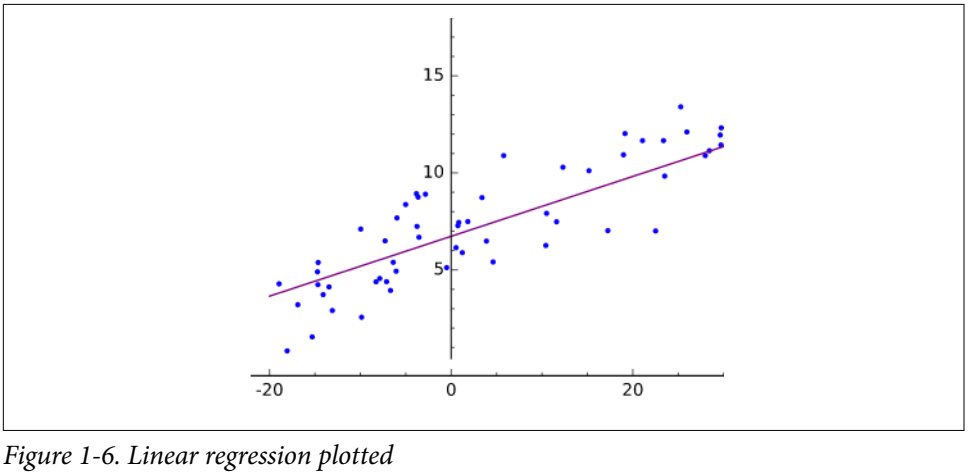

Figure 1-6. Linear regression plotted

Fitting is defining a function *f(x)* that produces *y*-values close to the measured or real-world *y* values. The line produced by $y = f(x)$ comes close to the scattered coordinates, the pairs of dependent and independent variables.

Relating the linear regression model

We can relate this function to the earlier equation, $Ax = b$, where A is the features (e.g., "weight" or "square footage") for all of the input examples that we want to model. Each input record is a row in the matrix A. The column vector b is the outcomes for all of the input records in the A matrix. Using an error function and an optimization method (e.g., SGD), we can find a set of x parameters such that we minimize the error across all of the predictions versus the true outcomes.

Using SGD, as we discussed earlier, we'd have three components to solve for our parameter vector x:

A hypothesis about the data
: The inner product of the parameter vector x and the input features (as displayed above)

A cost function
: Squared error (prediction – actual) of prediction

An update function
: The derivative of the squared error loss function (cost function)

While linear regression deals with straight lines, nonlinear curve fitting handles everything else, most notably curves that deal with x to higher exponents than 1. (That's why we hear machine learning sometimes described as "curve fitting.") An absolute fit would hit every dot on a scatterplot. Ironically, absolute fit is usually a very poor outcome, because it means your model has trained too perfectly on the training set, and has almost no predictive power beyond the data it has seen (e.g., does not generalize well), as we previously discussed.

Classification

Classification is modeling based on delineating classes of output based on some set of input features. If regression give us an outcome of "*how much*," classification gives us an outcome of "*what kind*." The dependent variable y is categorical rather than numerical.

The most basic form of classification is a binary classifier that only has a single output with two labels (two classes: 0 and 1, respectively). The output can also be a floating-point number between 0.0 and 1.0 to indicate a classification below absolute certainty. In this case, we need to determine a threshold (typically 0.5) at which we delineate between the two classes. These classes are often referred to as positive classi-

fication in the literature (e.g., 1.0) and then negative (e.g., 0.0) classifications. We'll talk more about this in "Evaluating Models" on page 36.

Examples of binary classification include:

- Classifying whether someone has a disease or not
- Classifying an email as spam or not spam
- Classifying a transaction as fraudulent or nominal

Beyond two labels, we can have classification models that have *N* labels for which we'd score each of the output labels, and then the label with the highest score is the output label. We'll discuss this further as we talk about neural networks with multiple outputs versus neural networks with a single output (binary classification). We'll also discuss classification more in this chapter when we talk about logistic regression and then dive into the full architecture of neural networks.

Recommendation

Recommendation is the process of suggesting items to users of a system based on similar other users or other items they have looked at before. One of the more famous variations of recommendation algorithms is called *Collaborative Filtering* made famous by Amazon.com.

Clustering

Clustering is an unsupervised learning technique that involves using a distance measure and iteratively moving similar items more closely together. At the end of the process, the items clustered most densely around *n* centroids are considered to be classified in that group. *K*-means clustering is one of the more famous variations of clustering in machine learning.

Underfitting and Overfitting

As we mentioned earlier, optimization algorithms first attempt to solve the problem of underfitting; that is, of taking a line that does not approximate the data well and making it approximate the data better. A straight line cutting across a curving scatterplot would be a good example of underfitting, as is illustrated in Figure 1-7.

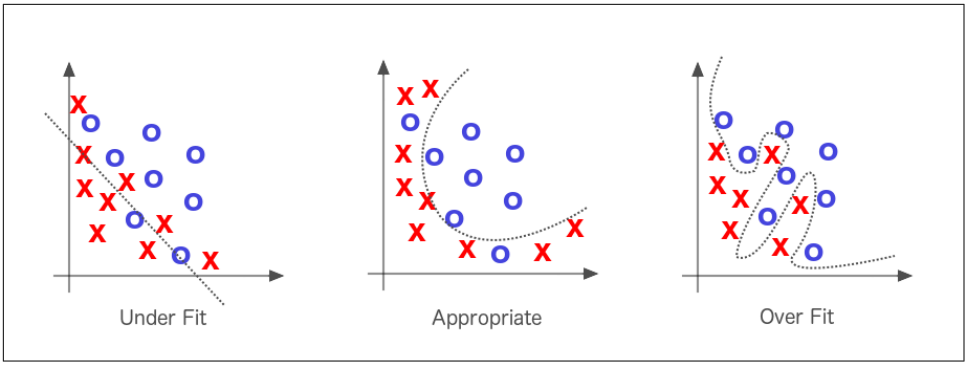

Figure 1-7. Underfitting and overfitting in machine learning

If the line fits the data too well, we have the opposite problem, called "overfitting." Solving underfitting is the priority, but much effort in machine learning is spent attempting not to overfit the line to the data. When we say a model overfits a dataset, we mean that it may have a low error rate for the training data, but it does not generalize well to the overall population of data in which we're interested.

Another way of explaining overfitting is by thinking about probable distributions of data. The training set of data that we're trying to draw a line through is just a sample of a larger unknown set, and the line we draw will need to fit the larger set equally well if it is to have any predictive power. We must assume, therefore, that our sample is loosely representative of a larger set.

Optimization

The aforementioned process of adjusting weights to produce more and more accurate guesses about the data is known as *parameter optimization*. You can think of this process as a scientific method. You formulate a hypothesis, test it against reality, and refine or replace that hypothesis again and again to better describe events in the world.

Every set of weights represents a specific hypothesis about what inputs mean; that is, how they relate to the meanings contained in one's labels. The weights represent conjectures about the correlations between networks' input and the target labels they seek to guess. All possible weights and their combinations can be described as the hypothesis space of this problem. Our attempt to formulate the best hypothesis is a matter of searching through that hypothesis space, and we do so by using error and optimization algorithms. The more input parameters we have, the larger the search space of our problem. Much of the work of learning is deciding which parameters to ignore and which to hear.

The Decision Boundary and Hyperplanes

When we mention the "decision boundary," we're talking about the *n*-dimensional hyperplane created by the parameter vector in linear modeling.

Fitting lines to data by gauging their cost (i.e., their distance from the ground-truth data points) is at the center of machine learning. The line should more or less fit the data, and it does so by minimizing the aggregate distance of all points from the line. You minimize the sum of the difference between the line at point x and the target point y to which it corresponds. In a three-dimensional space, you can imagine the error-scape of hills and valleys, and picture your algorithm as a blind hiker who feels for the slope. An optimization algorithm, like gradient descent, is what informs the hiker which direction is downhill so that she knows where to step.

The goal is to find the weights that minimize the difference between what your network predicts ($\hat{b}$, or the dot-product of A and x) and what your test set knows to be true (b), as we saw earlier in Figure 1-4. he parameter vector (x) above is where you would find the weights. The accuracy of a network is a function of its input and parameters, and the speed at which it becomes accurate is a function of its hyperparameters.

Hyperparameters

In machine learning, we have both model parameters and then we have parameters we tune to make networks train better and faster. These tuning parameters are called *hyperparameters*, and they deal with controlling optimization function and model selection during training with our learning algorithm.

Convergence

Convergence refers to an optimization algorithm finding values for a parameter vector that gives our optimization algorithm the smallest error possible across all training examples. The optimization algorithm is said to "converge" on the solution iteratively after it tries several different variations of the parameters.

Following are the three important functions at work in machine learning optimization:

Parameters
: Transform input to help determine the classifications a network infers

Loss function
: Gauges how well it classifies (minimizes error) at each step

Optimization function
: Guides it toward the points of least error

Now, let's take a closer look at one subclass of optimization called convex optimization.

Convex Optimization

In *convex optimization*, learning algorithms deal with convex cost functions. If the x-axis represents a single weight, and the y-axis represents that cost, the cost will descend as low as 0 at one point on the x-axis and rise exponentially on either side as the weight strays away from its ideal in two directions.

Figure 1-8 demonstrates that we also can turn the idea of a cost function upside down.

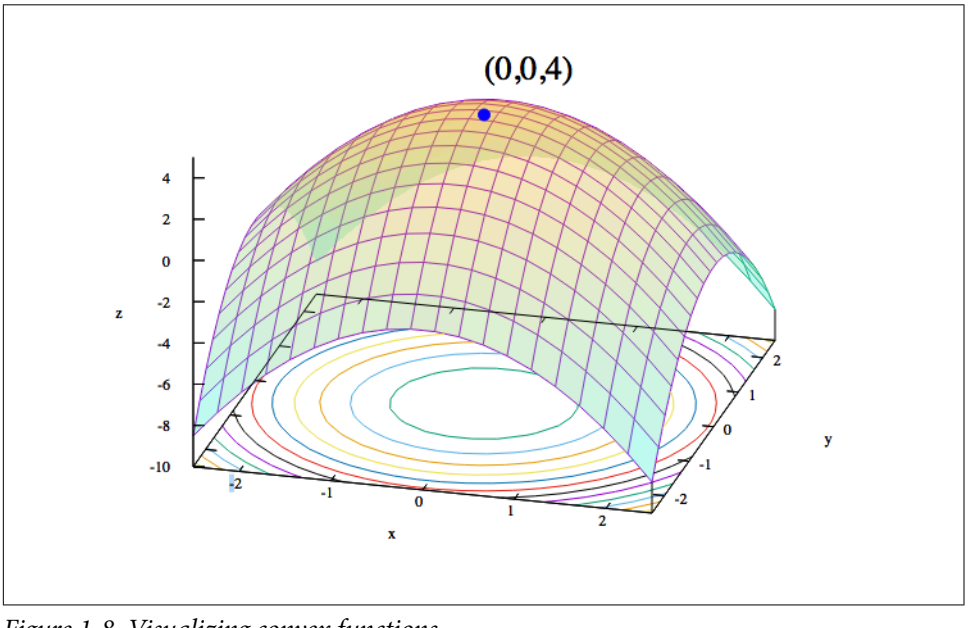

Figure 1-8. Visualizing convex functions

Another way to relate parameters to the data is with a maximum likelihood estimation, or MLE. The MLE traces a parabola whose edges point downward, with likelihood measured on the vertical axis, and a parameter on the horizontal. Each point on the parabola measures the likelihood of the data, given a certain set of parameters. The goal of MLE is to iterate over possible parameters until it finds the set that makes the given data most likely.

In a sense, maximum likelihood and minimum cost are two sides of the same coin. Calculating the cost function of two weights against the error (which puts us in a

three-dimensional space) will produce something that looks more like a sheet held at each corner and drooping convexly in the middle—a rather bowl-shaped function. The slopes of these convex curves allow our algorithm a hint as to what direction to take the next parameter step, as we'll see in the gradient descent optimization algorithm discussed next.

Gradient Descent

In gradient descent, we can imagine the quality of our network's predictions (as a function of the weight/parameter values) as a landscape. The hills represent locations (parameter values or weights) that give a lot of prediction error; valleys represent locations with less error. We choose one point on that landscape at which to place our initial weight. We then can select the initial weight based on domain knowledge (if we're training a network to classify a flower species we know petal length is important, but color isn't). Or, if we're letting the network do all the work, we might choose the initial weights randomly.

The purpose is to move that weight downhill, to areas of lower error, as quickly as possible. An optimization algorithm like gradient descent can sense the actual slope of the hills with regard to each weight; that is, it knows which direction is down. Gradient descent measures the slope (the change in error caused by a change in the weight) and takes the weight one step toward the bottom of the valley. It does so by taking a derivative of the loss function to produce the gradient. The gradient gives the algorithm the direction for the next step in the optimization algorithm, as depicted in Figure 1-9.

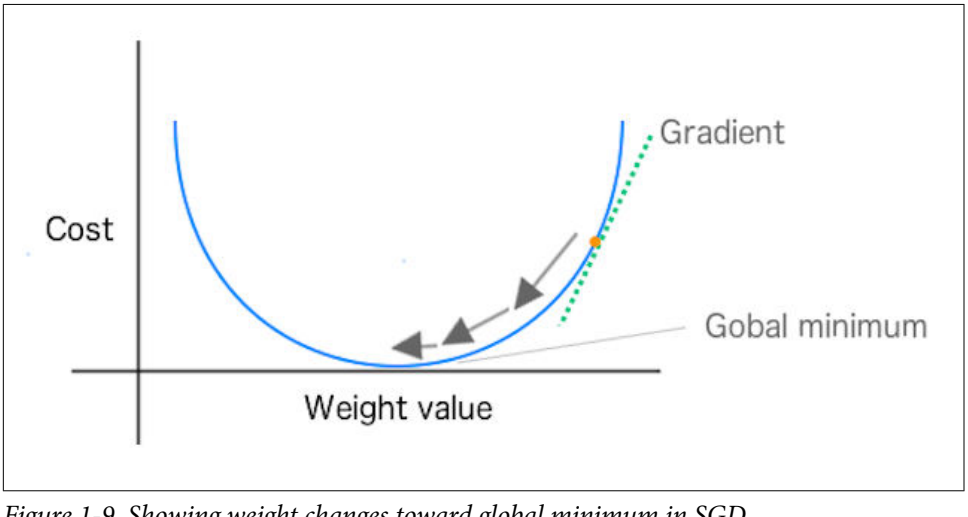

Figure 1-9. Showing weight changes toward global minimum in SGD

The derivative measures "rate of change" of a function. In convex optimization, we're looking for the point at which the derivative is equal to 0 for the function. This point is also known as the *stationary point* of the function or the *minimum point*. In optimization, we consider optimizing a function to be minimizing a function (outside of inverting the cost function).

What Is Gradient?

Gradient is defined as the generalization of the derivative of a function in one dimension to a function f in several dimensions. It is represented as a vector of n partial derivatives of the function f. It is useful in optimization in that the gradient points in the direction of the greatest rate of increase of the function for which the magnitude is the slope of the graph in that direction.

Gradient descent calculates the slope of the loss function by taking a derivative, which should be a familiar term from calculus. On a two-dimensional loss function, the derivative would simply be the tangent of any point on the parabola; that is, the change in y over the change in x, rise over run.

As we know from trigonometry, a tangent is just a ratio: the opposite side (which measures vertical change) over the adjacent side (which measures horizontal change) of a right triangle.

One definition of a curve is a line of constantly changing slope. The slope of each point on the curve is represented by the tangent line touching that point. Because slopes are derived from two points, how exactly does one find the slope of one point on a curve? We find the derivative by calculating the slope of a line between two points on the curve separated by a small distance and then slowing decreasing that distance until it approaches zero. In calculus, this is a *limit*.

This process of measuring loss and changing the weight by one step in the direction of less error is repeated until the weight arrives at a point beyond which it cannot go lower. It stops in the trough, the point of greatest accuracy. When using a convex loss function (typically in linear modeling), we see a loss function that has only a global minimum.

You can think of linear modeling in terms of three components to solve for our parameter vector x:

- A hypothesis about the data; for example, "the equation we use to make a prediction in the model."
- A cost function. Also called a loss function; for example, "sum of squared errors."
- An update function; we take the derivative of the loss function.

Our hypothesis is the combination of the learned parameters x and the input values (features) that gives us a classification or real valued (regression) output. The cost function tells us how far we are from the global minimum of the loss function, and we use the derivative of the loss function as the update function to change the parameter vector x.

Taking the derivative of the loss function indicates for each parameter in x the degree to which we need to adjust the parameter to get closer to the 0-point on the loss curves. We'll look more closely at these equations later on in this chapter when we show how they work for both linear regression and logistic regression (classification).

However, in other nonlinear problems we don't always get such a clean loss curve. The problem with these other nonlinear hypothetical landscapes is that there might be several valleys, and gradient descent's mechanism for taking the weight lower cannot know if it has reached the lowest valley or simply the lowest point in a higher valley, so to speak. The lowest point in the lowest valley is known as *the global minimum*, whereas the nadirs of all other valleys are known as *local minima*. If gradient descent reaches a local minimum, it is effectively trapped, and this is one drawback of the algorithm. We'll look at ways to overcome this issue in Chapter 6 when we examine hyperparameters and learning rate.

A second problem that gradient descent encounters is with non-normalized features. When we write "non-normalized features," we mean features that can be measured by very different scales. If you have one dimension measured in the millions, and another in decimals, gradient descent will have a difficult time finding the steepest slope to minimize error.

Dealing with Normalization

In Chapter 8 we take an extended look at methods of normalization in the context of vectorization and illustrate some ways to better deal with this issue.

Stochastic Gradient Descent

In gradient descent we'd calculate the overall loss across all of the training examples before calculating the gradient and updating the parameter vector. In SGD, we compute the gradient and parameter vector update after every training sample. This has been shown to speed up learning and also parallelizes well as we'll talk about more later in the book. SGD is an approximation of "full batch" gradient descent.

Mini-batch training and SGD

Another variant of SGD is to use more than a single training example to compute the gradient but less than the full training dataset. This variant is referred to as the *mini-*

batch size of training with SGD and has been shown to be more performant than using only single training instances. Applying mini-batch to stochastic gradient descent has also shown to lead to smoother convergence because the gradient is computed at each step it uses more training examples to compute the gradient.

As the mini-batch size increases the gradient computed is closer to the "true" gradient of the entire training set. This also gives us the advantage of better computational efficiency. If our mini-batch size is too small (e.g., 1 training record), we're not using hardware as effectively as we could, especially for situations such as GPUs. Conversely, making the mini-batch size larger (beyond a point) can be inefficient, as well, because we can produce the same gradient with less computational effort (in some cases) with regular gradient descent.

Quasi-Newton Optimization Methods

Quasi-Newton optimization methods are iterative algorithms that involve a series of "line searches." Their distinguishing feature with respect to other optimization methods is how they choose the search direction. These methods are discussed further in later chapters of the book.

The Jacobian and the Hessian

The Jacobian is an $m \times n$ matrix containing the first-order partial derivatives of vectors with respect to vectors.

The Hessian is the square matrix of second-order partial derivatives of a function. This matrix describes the local curvature of a function of many variables. We see the Hessian matrix used in large-scale optimization problems using Newton-type methods because they are the coefficients of the quadratic term of a local Taylor expansion. In practice, the Hessian can be computationally difficult to compute. We tend to see quasi-Newton algorithms used instead that approximate the Hessian. An example of this class of quasi-Newton optimization algorithm would be L-BFGS, which we cover in greater detail in Chapter 2.

We won't reference the Jacobian and the Hessian matricies much in this book, but we want the reader to be aware of them and their place in the wider scope of the machine learning landscape.

Generative Versus Discriminative Models

We can generate different types of output from a model depending on what type of model we set up. The two major types are *generative* models and *discriminative* models. Generative models understand how the data was created in order to generate a type of response or output. Discriminative models are not concerned with how the

data was created and simply give us a classification or category for a given input signal. Discriminative models focus on closely modeling the boundary between classes and can yield a more nuanced representation of this boundary than a generative model. Discriminative models are typically used for classification in machine learning.

A generative model learns the *joint* probability distribution $p(x, y)$, whereas a discriminative model learns the *conditional* probability distribution $p(y|x)$. The distribution $p(y|x)$ is the natural distribution for taking input x and producing an output (or classification) y, hence the name "discriminative model." With generative models learning the distribution $p(x,y)$, we see them used to generate likely output, given a certain input. Generative models are typically set up as probabilistic graphical models which capture the subtle relations in the data.

Logistic Regression

Logistic regression is a well-known type of classification in linear modeling. It works for both binary classification as well as for multiple labels in the form of multinomial logistic regression. Logistic regression is a regression model (technically) in which the dependent variable is categorical (e.g., "classification"). The binary logistic model is used to estimate the probability of a binary response based on a set of one or more input variables (independent variables or "features"). This output is the statistical probability of a category, given certain input predictors.

Similar to linear regression, we can express a logistic regression modeling problem in the form of $Ax = b$, where A is the feature (e.g., "weight" or "square footage") for all of the input examples we want to model. Each input record is a row in the matrix A, and the column vector b is the outcomes for all of the input records in the A matrix. Using a cost function and an optimization method, we can find a set of x parameters such that we minimize the error across all of the predictions versus the true outcomes.

Again, we'll use SGD to set up this optimization problem and we have three components to solve for our parameter vector x:

A hypothesis about the data

$$f(x) = \frac{1}{1 + e^{-\theta x}}$$

A cost function

"max likelihood estimation"

An update function

A derivative of the cost function

In this case, the input comprises independent variables (e.g., the input columns or "features"), whereas the output is the dependent variables (e.g., "label scores"). An

easy way to think about it is logistic regression function pairs input values with weights to determine whether an outcome is likely. Let's take a closer look at the logistic function.

The Logistic Function

In logistic regression we define the *logistic function* ("hypothesis") as follows:

$$f(x) = \frac{1}{1 + e^{-\theta x}}$$

This function is useful in logistic regression because it takes any input in the range of negative to positive infinity and maps it to output in the range of 0.0 to 1.0. This allows us to interpret the output value as a probability. Figure 1-10 shows a plot of the logistic function equation.

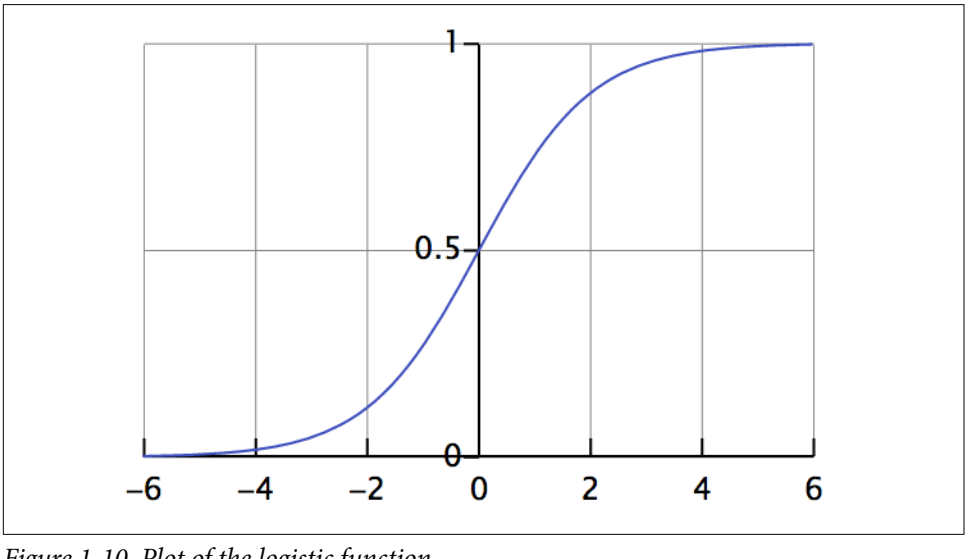

Figure 1-10. Plot of the logistic function

This function is known as a *continuous log-sigmoid function* with a range of 0.0 to 1.0. We'll see this function covered again later in "Activation Functions."

Understanding Logistic Regression Output

The logistic function is often denoted with the Greek letter sigma, or σ, because the relationship between x and y on a two-dimensional graph resembles an elongated, wind-blown "s" whose maximum and minimum asymptotically approach 1 and 0, respectively.

If y is a function of x, and that function is sigmoidal or logistic, the more x increases, the closer we come to 1/1, because e to the power of an infinitely large negative number approaches zero; in contrast, the more x decreases below zero, the more the expression $(1 + e^{-\theta x})$ grows, shrinking the entire quotient. Because $(1 + e^{-\theta x})$ is in the denominator, the larger it becomes, the closer the quotient itself comes to zero.

With logistic regression, $f(x)$ represents the probability that y equals 1 (i.e., is true) given each input x. If we are attempting to estimate the probability that an email is spam, and $f(x)$ happens to equal 0.6, we could paraphrase that by saying y has a 60 percent of being 1, or the email has a 60 percent chance of being spam, given the input. If we define machine learning as a method to infer unknown outputs from known inputs, the parameter vector x in a logistic regression model determines the strength and certainty of our deductions.

The Logit Transformation

The logit function is the inverse of the logistic function ("logistic transform").

Evaluating Models

Evaluating models is the process of understanding how well they give the correct classification and then measuring the value of the prediction in a certain context. Sometimes, we only care how often a model gets any prediction correct; other times, it's important that the model gets a certain type of prediction correct more often than the others. In this section, we cover topics like bad positives, harmless negatives, unbalanced classes, and unequal costs for predictions. Let's take a look at the basic tool for evaluating models: the *confusion matrix*.

The Confusion Matrix

The confusion matrix (see Figure 1-11)—also called a *table of confusion*—is a table of rows and columns that represents the predictions and the actual outcomes (labels) for a classifier. We use this table to better understand how well the model or classifier is performing based on giving the correct answer at the appropriate time.

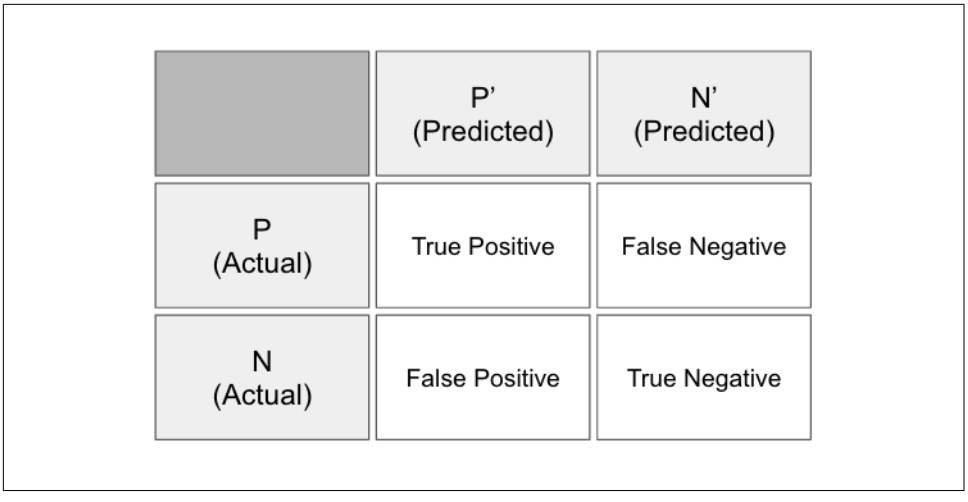

Figure 1-11. The confusion matrix

We measure these answers by counting the number of the following:

- True positives
  - — Positive prediction
  - — Label was positive
- False positives
  - — Positive prediction
  - — Label was negative
- True negatives
  - — Negative prediction
  - — Label was negative
- False negatives
  - — Negative prediction
  - — Label was positive

In traditional statistics a false positive is also known as "type I error" and a false negative is known as a "type II error." By tracking these metrics, we can achieve a more detailed analysis on the performance of the model beyond the basic percentage of guesses that were correct. We can calculate different evaluations of the model based on combinations of the aforementioned four counts in the confusion matrix, as shown here:

```
Accuracy:  0.94
Precision: 0.8662
Recall:    0.8955
F1 Score:  0.8806
```

In the preceding example, we can see four different common measures in the evaluation of machine learning models. We'll cover each of them shortly, but for now, let's begin with the basics of evaluating model sensitivity versus model specificity.

Sensitivity versus specificity

Sensitivity and *specificity* are two different measures of a binary classification model. The true positive rate measures how often we classify an input record as the positive class and its the correct classification. This also is called sensitivity, or recall; an example would be classifying a patient as having a condition who was actually sick. Sensitivity quantifies how well the model avoids false negatives.

Sensitivity = TP / (TP + FN)

If our model was to classify a patient from the previous example as not having the condition and she actually did not have the condition then this would be considered a true negative (also called specificity). Specificity quantifies how well the model avoids false positives.

Specificity = TN / (TN + FP)

Many times we need to evaluate the trade-off between sensitivity and specificity. An example would be to have a model that detects serious illness in patients more frequently because of the high cost of misdiagnosing a truly sick patient. We'd consider this model to have low specificity. A serious sickness could be a danger to the patient's life and to the others around him, so our model would be considered to have a high sensitivity to this situation and its implications. In a perfect world, our model would be 100 percent sensitive (i.e., all sick are detected) and 100 percent specific (i.e., no one who is not sick is classified as sick).

Accuracy

Accuracy is the degree of closeness of measurements of a quantity to that quantity's true value.

Accuracy = (TP + TN) / (TP + FP + FN + TN)

Accuracy can be misleading in the quality of the model when the class imbalance is high. If we simply classify everything as the larger class, our model will automatically get a large number of its guesses correct and provide us with a high accuracy score yet misleading indication of value based on a real application of the model (e.g., it will never predict the smaller class or rare event).

Precision

The degree to which repeated measurements under the same conditions give us the same results is called precision in the context of science and statistics. Precision is also known as the *positive prediction value*. Although sometimes used interchangeably with "accuracy" in colloquial use, the terms are defined differently in the frame of the scientific method.

$$\text{Precision} = TP / (TP + FP)$$

A measurement can be accurate yet not precise, not accurate but still precise, neither accurate nor precise, or both accurate and precise. We consider a measurement to be valid if it is both accurate and precise.

Recall

This is the same thing as sensitivity and is also known as the *true positive rate* or the *hit rate*.

F1

In binary classification we consider the F1 score (or F-score, F-measure) to be a measure of a model's accuracy. The F1 score is the harmonic mean of both the precision and recall measures (described previously) into a single score, as defined here:

$$F1 = 2TP / (2TP + FP + FN)$$

We see scores for F1 between 0.0 and 1.0, where 0.0 is the worst score and 1.0 is the best score we'd like to see. The F1 score is typically used in information retrieval to see how well a model retrieves relevant results. In machine learning, we see the F1 score used as an overall score on how well our model is performing.

Context and interpreting scores

Context can play a role in how we evaluate our model and dictate when we use different types of scores, as described previously in this section. Class imbalance can play a large role in dictating choice of evaluation score, and in many datasets, we'll find that the classes or label counts are not well balanced. Here are some typical domains in which we see this:

- Web click prediction
- ICU mortality prediction
- Fraud detection

In these contexts, an overall "percent correct" score can be misleading to the overall value, in practical terms, of the model. An example of this would be the PhysioNet Challenge dataset from 2012.

The goal of the challenge was to "predict in-hospital *mortality* with the greatest accuracy using a binary classifier." The difficulty and challenge in modeling this dataset is that predicting that a patient will live is the easy part because the bulk of examples in the dataset have outcomes in which the patient does live. Predicting death accurately in this scenario is the goal; this is where the model has the most value in the context of being clinically relevant in the real world. In this competition, the scores were calculate as follows:

$$\text{Score} = \text{MIN}(\text{Precision}, \text{Recall})$$

This was set up such that it kept the contestants focused on not just predicting that the patient would live most of the time and get a nice F1 score, but focus on predicting when the patient would die (keeping the focus on being clinically relevant). This is a great example of how context can change how we evaluate our models.

Methods to Handle Class Imbalance

In Chapter 6 we illustrate practical ways to deal with class imbalance. We take a closer look at the different facets of class imbalance and error distributions in the context of classification and regression.

Building an Understanding of Machine Learning

In this chapter, we introduced the core concepts needed for practicing machine learning. We looked at the core mathematical concepts of modeling based around the equation:

$$Ax = b$$

We also looked at the core ideas of getting features into the matrix A, ways to change the parameter vector x, and setting the outcomes in the vector b. We illustrated some basic ways to change the parameter vector x to minimize the score (or "loss") of the objective function.

As we move forward in this book, we'll continue to expand on these key concepts. We'll see how neural networks and deep learning are based on these fundamentals but add more complex ways to create the A matrix, change the x parameter vector through optimization methods, and measure loss during training. Let's now move on to Chapter 2, where we build further on these concepts with the basics of neural networks.

CHAPTER 2

Fundamentals of Deep Networks

Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!

—The Red Queen, *Through the Looking Glass*

Defining Deep Learning

In the Chapter 2 we set up the foundations of machine learning and neural networks. In this chapter we'll build on these foundations to give you the core concepts of deep networks. This will help build your understanding of what is going on in different network architectures as we progress into the specific architectures in Chapter 4 and then the practical examples in Chapter 5. Let's begin by restating our definitions of both deep learning and deep networks.

What Is Deep Learning?

Revisiting our definition of deep learning from Chapter 1, the facets that differentiate deep learning networks in general from "canonical" feed-forward multilayer networks are as follows:

- More neurons than previous networks
- More complex ways of connecting layers
- "Cambrian explosion" of computing power to train
- Automatic feature extraction

When we say "more neurons," we mean that the neuron count has risen over the years to express more complex models. Layers also have evolved from each layer being fully connected in multilayer networks to locally connected patches of neurons between layers in Convolutional Neural Networks (CNNs) and recurrent connections

to the same neuron in Recurrent Neural Networks (in addition to the connections from the previous layer).

More connections means that our networks have more parameters to optimize, and this required the explosion in computing power that occurred over the past 20 years. All of these advances provided the foundation to build next-generation neural networks capable of extracting features for themselves in a more intelligent fashion. This allowed deep networks to model more complex problem spaces (e.g., image recognition advances) than previously possible. As industry demands are ever changing and ever reaching, the capabilities of neural networks have had to charge forward. The Red Queen[1] would have it no other way.

Defining deep networks

To further provide color to our definition of deep learning, here we define the four major architectures of deep networks:

- Unsupervised Pretrained Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

There is continuous research in the domain of neural networks, but for the purposes of this book, we'll focus on these four architectures. These architectures have evolved over the past 20 years. Let's take a quick look at some of the highlights, continuing our history lesson that began in Chapter 1 on the history of feed-forward multilayer neural networks.

Deep Reinforcement Learning

Reinforcement learning is defined in Sutton's book as follows:

> Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem.

It goes on to state that any method suitable to solving that problem can be considered to be a reinforcement learning method. In reinforcement learning, we do not tell the learner which actions to take, but let the agent discover the actions yielding the best reward by experimenting in a simulation.

In reinforcement learning, the agent begins with no trained model of the environment, and the utility function is synonymous with the reward or objective our agent is after. The training system gives the agent input from the environment and rewards

1 Referring to Lewis Carroll's Red Queen character from the book *Through the Looking Glass*; she has to keep running to stay in the same place.

the agent when the outcome (of the cycle, or *frame*) of the simulation (or *game*) being played is positive. Many times, actions will affect not only the immediate reward, but also future rewards. The mechanics of trial-and-error and delayed reward are key features of reinforcement learning.

Deep reinforcement learning is a variant of reinforcement learning in which a neural network is used as the universal function's approximator. The drawback to this approach is that the behavior of a neural network is not bounded, and the proof of convergence doesn't hold anymore. However, using neural networks as the universal function approximator shows good results despite this.

In 2013 a paper was published by the DeepMind team at the NIPS 2013 Deep Learning Workshop on playing ATARI games with Deep Q Learning.[2] In this paper, the authors used a standard algorithm (Q Learning with function approximation). Their function approximator for this algorithm was a CNN. They demonstrated an agent playing Atari 2600 games based on using the pixels on the screen as input and a CNN as the internal model.

The computer/agent playing the ATARI game was positively rewarded when the outcome of the game was positive based on the actions performed. The algorithm was able to learn some of the games to a level where it performed better than humans.

Deep reinforcement learning has risen in popularity over the course of production of this book, and hopefully we'll see it included in a future edition. For now we'll direct you to an example in Appendix B.

Evolutionary progress and resurgence

When we last left off in Chapter 2, neural networks had entered a "winter period" in the mid-1980s when the promise of AI fell short of what it could deliver. As happens many times when promising technology falls into the Trough of Disillusionment (Figure 2-1), there were many researchers still doing important work in the realm of neural networks.

2 Mnih et al. 2013. "Human-level control through deep reinforcement learning."

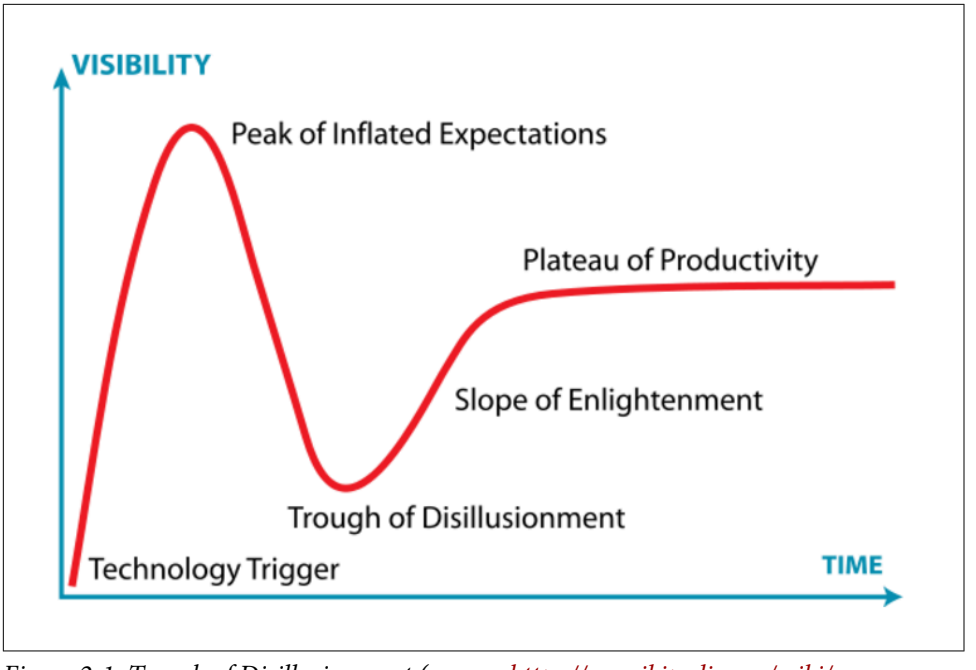

Figure 2-1. Trough of Disillusionment (source: https://en.wikipedia.org/wiki/Hype_cycle)

One important development in neural networks was Yann LeCun's work at AT&T Bell Labs on optical character recognition.[3] His lab was focused on check image recognition for the financial services sector. Through this work, LeCun and his team developed the concept of the biologically inspired model of image recognition we know today as CNN. This eventually led to the creation of the MNIST handwriting benchmark (we cover this more later in the chapter) and a progressive number of record accuracy marks achieved by deep learning.

Better Labeled Data

Another contributing factor to the evolution and success of deep networks was the creation of better and larger labeled datasets such as MNIST and ImageNet.

Advances in modeling sequential data with recurrent neural networks appeared in the late-1980s and early 1990s by researchers such as Sepp Hochreiter. As time went on, the research community created better artificial neuron variants (e.g., Long Short-

3 LeCun et al. 1998. "Gradient-based learning applied to document recognition."

Term Memory [LSTM] Memory Cell and Memory Cell with Forget Gate) over the course of the late 1990s. The stage for a neural network resurgence was being set quietly in research labs around the world.

During the 2000s, researchers and industry applications began to progressively apply these advances in products such as the following:

- Self-driving cars
- Google Translate
- Amazon Echo
- AlphaGo

Self-driving cars in the 2006 Darpa Grand Challenge used many techniques beyond just deep learning. The top teams (Stanford and Carnegie Mellon University) were able to take advantage of the big improvements of image processing.

Advances in Computer Vision

In 2012 Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton developed a "large, deep convolutional neural network" that won the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge).

AlexNet[4] was hailed as an advancement in computer vision and some credit it specifically with kicking off the deep learning craze. However, it was largely a scaled-up (e.g., deeper and wider) variant of the CNNs from the 1990s. The recent advances in computer vision were driven less by recent algorithm advances and more by better compute, data, and infrastructure.

Better image analysis allowed the planning systems in the cars to better choose paths through uncertain terrain and avoid obstacles more safely. Other advances in deep learning allowed models to more accurately translate and recognize audio data, driving value in the Google Translate and Amazon Echo line of products. Most recently, we've seen another complex game fall at the master level when the AlphaGo system beat the 9-dan professional Go player Lee Sedol.

Big advances in what machine learning can accomplish are not always easy to see. Public recognition for these advances many times is the culmination of many different lines of work that is exhibited in high-profile demonstrations such as The Darpa Grand Challenge or Watson beating Ken Jennings in Jeopardy. However, behind the scenes, the underpinnings for these advances change slowly but constantly. Just like

4 Krizhevsky, Sutskever, and Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks."

the changing of the seasons, we don't always notice these changes in our daily lives until they've crossed some threshold.

In the near future, we'll continue to see deep learning being applied in unique and innovative ways. This application will be more of the latent intelligence variety (e.g., recommendations or voice recognition) coupled with pragmatic engineering to make them useful in everyday aspects of our lives. What we're unlikely to see (in the near term, at least) are out of control malevolent artificial agents blowing us out of airlocks at inopportune times (think HAL 9000 from *2001: A Space Odyssey*).

HAL 9000

HAL 9000 is the fictional computer who controls the systems of the *Discovery One spaceship* in Arthur C. Clarke's book *2001: A Space Odyssey*. HAL stands for Heuristically programmed ALgorithmic computer. HAL is represented largely in the film as a camera lens with a glowing red dot and is accessed through a conversational voice recognition system. In the film, HAL concludes that it must kill off the crew of the *Discovery One* to complete its mission successfully.

> Dave: Open the pod bay doors, HAL.
> HAL: I'm sorry, Dave. I'm afraid I can't do that.

Deep learning continues to push the field forward in many domains and on many core machine learning problems. Here are just a few of the benchmark records deep learning has achieved in the last few years:

- Text-to-speech synthesis (Fan et al., Microsoft, Interspeech 2014)
- Language identification (Gonzalez-Dominguez et al., Google, Interspeech 2014)
- Large vocabulary speech recognition (Sak et al., Google, Interspeech 2014)
- Prosody contour prediction (Fernandez et al., IBM, Interspeech 2014)
- Medium vocabulary speech recognition (Geiger et al., Interspeech 2014)
- English-to-French translation (Sutskever et al., Google, NIPS 2014)
- Audio onset detection (Marchi et al., ICASSP 2014)
- Social signal classification (Brueckner & Schulter, ICASSP 2014)
- Arabic handwriting recognition (Bluche et al., DAS 2014)
- TIMIT phoneme recognition (Graves et al., ICASSP 2013)
- Optical character recognition (Breuel et al., ICDAR 2013)
- Image caption generation (Vinyals et al., Google, 2014)
- Video-to-textual description (Donahue et al., 2014)
- Syntactic parsing for natural language processing (Vinyals et al., Google, 2014)
- Photo-real talking heads (Soong and Wang, Microsoft, 2014)

Based on these accomplishments, we can easily project deep learning to impact many applications over the next decade. Some of the more impressive demonstrations of applied deep learning include the following:

- Automated image sharpening
- Automating image upscaling
- WaveNet: generating human speech that can imitate anyone's voice
- WaveNet: generating believable classical music
- Speech reconstruction from silent video
- Generating fonts
- Image autofill for missing regions
- Automated image captioning (see also: *https://github.com/karpathy/neuraltalk2*)
- Turning hand-drawn doodles into stylized artwork

We probably won't realize all of the major commercial applications until they're right in front of our faces. Understanding the advances in deep network architecture is important in order to understand application ideas going forward.

Advances in network architecture

As research pressed the state of the art forward from multilayer feed-forward networks toward newer architectures like CNNs and Recurrent Neural Networks, the discipline saw changes in how layers were set up, how neurons were constructed, and how we connected layers. Network architectures evolved to take advantage of specific types of input data.

Advances in layer types. Layers became more varied with the different types of architectures. Deep Belief Networks (DBNs) demonstrated success with using Restricted Boltzmann Machines (RBMs) as layers in pretraining to build features. CNNs used new and different types of activation functions in layers and changed how we connected layers (from fully connected to locally connected patches). Recurrent Neural Networks explored the use of connections that better modeled the time domain in time-series data.

Advances in neuron types. Recurrent Neural Networks specifically created advancements in the types of neurons (or units) applied in the work around LSTM networks. They introduced new units specific to Recurrent Neural Networks such as the LSTM Memory Cell and Gated Recurrent Units (GRUs).

Hybrid architectures. Continuing the theme of matching input data to architecture type, we have seen hybrid architectures emerge for types of data that has both a time domain and image data involved. For instance, classifying objects in video has been successfully demonstrated by combining layers from both CNNs and Recurrent Neural Networks into a single hybrid network. Hybrid neural network architectures can allow us to take advantage of the best of both worlds in some cases.

From feature engineering to automated feature learning

Although deep networks might have innovated with new units and layers for their internals, they are still fundamentally capped with a discriminatory classifier at the end with the constructed features as input. Automating feature extraction is a common theme among the various architectures. Each architecture does feature construction differently and is specialized such that it is better at certain types of input than others. Yann LeCun hit on this theme describing deep learning when he said it was: "machines that learn to represent the world."

Geoffrey Hinton talks about this theme in DBNs when he explains how RBMs are used to decompose the data into higher-order features.[5]

Categorizing DBNs

For the purposes of this book, we place DBNs (and autoencoders) in the UPN group of deep networks.

Staying with the image classification theme, we can use the example of face detection. Raw image data of faces as input have issues with how the face is oriented, lighting of the photo, and position of the key features of the face. The key features we'd normally associate with a face are things like the edge of the face, edges of specific features like eyes and nose, and then subtle features we don't consistently see like dimples.

Feature engineering. Handcrafting features has been a hallmark of machine learning for a long time. Practitioners who win competitions in machine learning often study the dataset thoroughly and use many arcane tricks to make the learning process as simple as possible for their learning algorithm. These datasets are often columnar/tabular text data and we can apply domain knowledge to specific columns so feature creation is more direct.

If we think back to Chapter 1 and how we modeled the input data as the A matrix in the equation $Ax = b$, we can see how we had to hand-code the values from the data into those specific columns of A. These handcrafted features tend to produce highly accurate models but take a lot of time and experience to produce. From a knowledge representation perspective, it's like reading a poorly written book versus a book that is well-written and easy to read. The former takes us a lot longer to read and we need to spend more energy to get the same out of it as the latter.

Image classification is an interesting example because handcrafting image features is more difficult than creating features for tabular data. The information in images is

5 Hinton, Osindero, and Teh. 2006. "A Fast Learning Algorithm for Deep Belief Nets."

not constrained to stay in the same column and can be influenced by lighting, angle, and other issues. Feature extraction and creation for images needed a new approach, which in some part drove the evolution of CNNs.

Feature learning. Coming back to our face-detection example, a nose can be located in any set of pixels in an image as opposed to our bank balance always being located in a specific column in tabular data. With CNNs, we train the network to understand the edge of the nose and then the general shape of the nose from lower-level "nose-edge" features. The first layers in the network might pick up those nose-edge features and then pass them on to later layers in the network as larger *feature maps*.

These more granular patches of feature maps eventually are combined into a "face" feature at the latter layers of the CNN. This allows a CNN to take on a task that has been attempted many times before ("Is this a face?") yet pose the question in a simpler way that takes less energy to answer in a more accurate way.

Automated Feature Learning with Complex Data

Taking complex raw data and creating higher-order features automatically in order to make a simpler classification (or regression) output is a hallmark of deep learning.

As you progress through this book, you'll get a better sense of how to match input data types to deep network architectures and how to set up these architectures to best model the underlying dataset.

Generative modeling

Generative modeling is not a new concept, but the level to which deep networks have taken it has begun to rival human creativity. From generating art to generating music to even writing beer reviews, we see deep learning applied in creative ways every day. Recent variants of generative modeling to note include the following:

- Inceptionism
- Modeling artistic style
- Generative Adversarial Networks
- Recurrent Neural Networks

Let's quickly review each of these.

Inceptionism. Inceptionism is a technique in which a trained convolutional network is taken with its layers in reverse order and given an input image coupled with a prior constraint. The images are modified iteratively to enhance the output in a manner that could be described as "hallucinative." In examples for which the input involves images of the sky, we might see fish faces appear in clouds of the output image. This

line of research from Google has shown discriminatory neural network models contain considerable information to generate images.

Modeling artistic style. Variants of convolutional networks have shown to learn the style of specific painters and then generate a new image in this style of arbitrary photographs. Figure 2-2 (which you saw earlier in Chapter 1) shows the amazing results. Imagine having your family photo painted by Vincent van Gogh. (By the time this book is published, this will probably be a Snapchat filter, so you won't have to wait that long.)

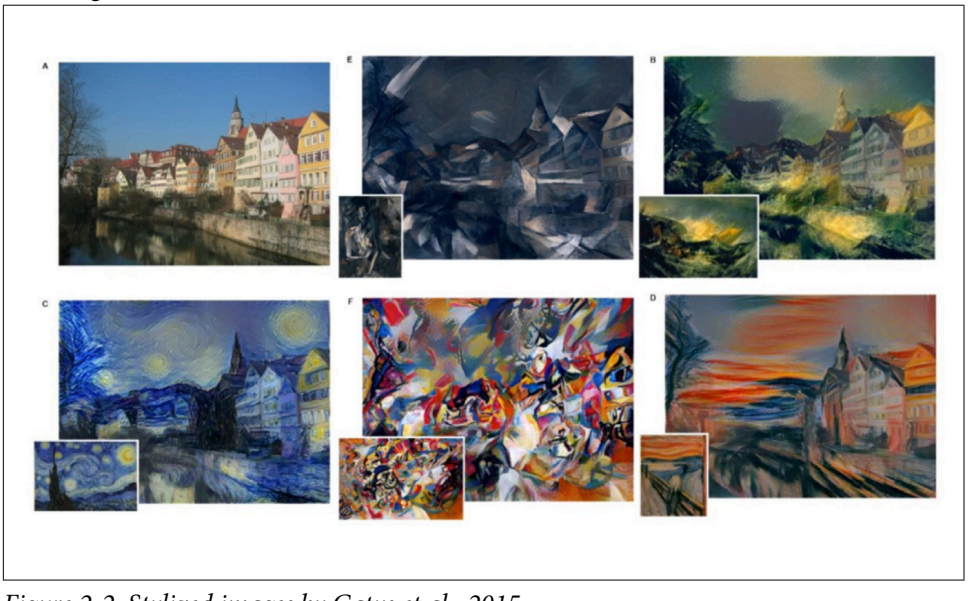

Figure 2-2. Stylized images by Gatys et al., 2015

In 2015, Gatys et al. published a paper titled "A Neural Algorithm of Artistic Style"[6] in which they separate the style and the content of a painting. The CNN extracts the artist's style into the network's parameters, which can later be applied to arbitrary images to be rendered in the same style.

GANs. The generative visual output of a GAN[7] can best be described as synthesizing novel images by modeling the distributions of input data seen by the network. We cover GANs in more depth in Chapter 4.

Recurrent Neural Networks. Recurrent Neural Networks have been shown to model sequences of characters and generate new sequences that are lucidly coherent. In

6 Gatys et al., 2015. "A Neural Algorithm of Artistic Style."

7 Goodfellow et al. 2014. "Generative Adversarial Networks."

Chapter 5, we take a look at an example of Recurrent Neural Networks in which we generate new lines of Shakespeare by modeling all of Shakespeare's other works.

Another interesting application of Recurrent Neural Networks is the work by Lipton and Elkan in which the network models proper nouns like "Coors Light" and other aspects of beer jargon. The generated beer reviews can be guided with hints (e.g., "give me a 3-star review of a German lager") and are impressive. Here's a sample beer review generated by the program:

> On tap at the brewpub. A nice dark red color with a nice head that left a lot of lace on the glass. Aroma is of raspberries and chocolate. Not much depth to speak of despite consisting of raspberries. The bourbon is pretty subtle as well. I really don't know that find a flavor this beer tastes like. I would prefer a little more carbonization to come through. It's pretty drinkable, but I wouldn't mind if this beer was available.[8]

The Tao of deep learning

There is a lot of marketing noise and hype in the realm of deep learning today, some of it justifiably so. However, deep learning is still trying to answer the same fundamental machine learning questions like: "Is this image a face?" The difference is that deep learning has taken the previous generation's neural network techniques and added advanced automated feature construction to make computationally difficult questions on complex data easier to answer.

When you use deep learning as a practitioner, the best way to take advantage of this power is to match the input data to the appropriate deep network architecture. If you do this, you can apply deep learning successfully in new and interesting ways. If you don't, you won't add any new modeling power beyond basic techniques like logistic regression. The remainder of this book is dedicated to giving you, as practitioner, the skills and context necessary to make these decisions and use deep learning well.

Organization of This Chapter

In this chapter, we build on the concepts from Chapter 1 and dig further into specific architectures for deep networks. We'll differentiate the architectures and break down how their components evolved differently, providing color on how this better extracts features from certain types of data. We close the chapter with some discussion of the practicality of deep learning and alleviate some misconceptions surrounding the domain today. With that, let's continue our discussion of the architecture components relevant to deep networks.

8 Source: IEEE Spectrum.

Common Architectural Principles of Deep Networks

Before we get into the specific architectures of the major deep networks, let's extend our understanding of the core components. First, we'll reexamine the core components again as follows and extend their coverage for the purposes of understanding deep networks:

- Parameters
- Layers
- Activation functions
- Loss functions
- Optimization methods
- Hyperparameters

Next, we'll take these concepts and build on them to better understand the building block networks of deep networks, such as the following:

- RBMs
- Autoencoders

We'll then continue to build on these ideas by reviewing these specific deep network architectures:

- UPNs
- CNNs
- Recurrent neural networks
- Recursive neural networks

As we work our way through this chapter, we'll also drop references to how DL4J implements certain aspects of deep networks. For now, let's continue our review of parameters to better understand how they are extended for deep networks.

Parameters

We learned in Chapter 1 that parameters relate to the x parameter vector in the equation $Ax = b$ in basic machine learning. Parameters in neural networks relate directly to the weights on the connections in the network. In Figure 1-4 (introduced first in Chapter 1) we can see the parameter vector represented by the x column vector. We take the dot product of the matrix A and the parameter vector x to get our current output column vector b. The closer our outcome vector b is to the actual values in the training data, the better our model is. We use methods of optimization such as gradient descent to find good values for the parameter vector to minimize loss across our training dataset.

In deep networks, we still have a parameter vector representing the connection in the network model we're trying to optimize. The biggest change in deep networks with respect to parameters is how the layers are connected in the different architectures. In DBNs, we see two parallel sets of feed-forward connections with two separate networks. One network's layers are composed of RBMs (subnetworks in their own right, which we'll review later in the chapter) used to extract features for the other network. The other network in a DBN is a regular feed-forward multilayer neural network, which uses the features extracted from the RBMs–layer network to initialize its weights. This is just one example of many that we'll see over the course of this chapter in how parameters/weights are specialized in different deep network architectures.

Parameters and NDArrays

In terms of working with the core linear algebra of deep networks, DL4J relies on the ND4J library to represent these linear algebra primitives. NDArrays and linear algebra are key to working with neural networks in DL4J.

Layers

In Chapter 1, we learned how input, hidden, and output layers define feed-forward neural networks. In Chapter 2, we further expanded this architecture with more types of layers and discussed how they relate to specific architectures of deep networks. Layers also can be represented by subnetworks in certain architectures, as well. In the previous section, we used the example of DBNs having layers composed of RBMs.

Layers are a fundamental architectural unit in deep networks. In DL4J we customize a layer by changing the type of activation function it uses (or subnetwork type in the case of RBMs). We'll also look at how you can use combinations of layers to achieve a goal (e.g., classification or regression). Finally, we'll also explore how each type of layer requires different hyperparameters (specific to the architecture) to get our network to learn initially. Further hyperparameter tuning can then be beneficial through reducing overfitting.

Activation Functions

In Chapter 1, we reviewed the basic activation functions used in feed-forward neural networks. In this chapter, we begin to illustrate how activation functions are used in specific architectures to drive feature extraction. The higher-order features learned from the data in deep networks are a nonlinear transform applied to the output of the previous layer. This allows the network to learn patterns in the data within a constrained space.

Activation functions for general architecture

Depending on the activation function you pick, you will find that some objective functions are more appropriate for different kinds of data (e.g., dense versus sparse). We group these design decisions for network architecture into two main areas across all architectures:

- Hidden layers
- Output layers

Hidden layers are concerned with extracting progressively higher-order features from the raw data. Depending on the architecture we're working with, we tend to use certain subsets of layer activation functions. As we work through this chapter, we'll illustrate these patterns more across DBNs, CNNs, and Recurrent Neural Networks. In Chapter 4, we take a deeper look at the impacts of different activation functions on different network architectures in the context of tuning deep networks.

A Note About Input Layers

Typically for the input layer, we want to pass on the raw input vector features so that in practice we don't express an activation function for the input layer.

Hidden layer activation functions. Commonly used functions include:

- Sigmoid
- Tanh
- Hard tanh
- Rectified linear unit (ReLU) (and its variants)

A more continuous distribution of input data is generally best modeled with a ReLU activation function. Optionally, we'd suggest using the tanh activation function (if the network isn't very deep) in the event that the ReLU did not achieve good results (with the caveat that there could be other hyperparameter-related issues with the network).

Sigmoid Activation Functions in Practice

In recent years, we've seen the sigmoid activation function fall out of favor for hidden layers in practice and research.

As we progress further in this book, we'll see how these activation functions tend to be arranged in the different architectures.

The Evolution of Activation Functions in Practice

We're also seeing an entire family of ReLUs emerge in deep learning research, such as the "leaky ReLU." You can read more on this topic in Chapter 6.

Output layer for regression. This design decision is motivated by what type of answer we expect our model to output. If we want to output a single real-valued number from our model, we'll want to use a linear activation function.

Output layer for binary classification. In this case, we'd use a sigmoid output layer with a single neuron to give us a real value in the range of 0.0 to 1.0 (excluding those values) for the single class. This real-valued output is typically interpreted as a probability distribution.

Output layer for multiclass classification. If we have a multiclass modeling problem yet we only care about the best score across these classes, we'd use a softmax output layer with an `arg-max()` function to get the highest score of all the classes. The softmax output layer gives us a probability distribution over all the classes.

Getting Multiple Classifications

If we want to get multiple classifications per output (e.g., person + car), we do not want softmax as an output layer. Instead, we'd use the sigmoid output layer with *n* number of neurons, giving us a probability distribution (0.0 to 1.0) for every class independently.

Loss Functions

In Chapter 2 we introduced loss functions and their role in machine learning. Loss functions quantify the agreement between the predicted output (or label) and the ground truth output. We use loss functions to determine the penalty for an incorrect classification of an input vector. So far, we've introduced the following loss functions:

- Squared loss
- Logistic loss
- Hinge loss
- Negative log likelihood

Previously we described loss functions as falling into one of three camps:

- Regression
- Classification
- Reconstruction

We covered the first two types in Chapter 1. The third, reconstruction, is involved in unsupervised feature extraction and is an important reason why deep learning networks have achieved their record-breaking accuracy. In certain architectures of deep networks reconstruction loss functions help the network extract features more effectively when paired with the appropriate activation function. An example of this would be using the multiclass cross-entropy as a loss function in a layer with a softmax activation function for classification output. We cover a specialized loss function in the next section.

Reconstruction cross-entropy

With the reconstruction entropy loss function, we first apply "Gaussian noise" (a kind of statistical white noise), and then the loss function punishes the network for any result that is less similar to the original input data. This feedback drives the network to learn different features in an attempt to reconstruct the input more effectively and minimize error. In deep learning, reconstruction entropy loss is used for feature engineering in the pretrain phase that involves RBMs.

Optimization Algorithms

Training a model in machine learning involves finding the best set of values for the parameter vector of the model. We can think of machine learning as an optimization problem in which we minimize the loss function with respect to the parameters of our prediction function (based on our model).

Defining "Best" in Terms of the Loss Function

In optimization algorithms, we define "best set of values" for the parameter vector as the values with the lowest loss function value.

In Chapter 1, we introduced the basic concepts of optimization, gradient descent, and parameter vectors. In this section, we take a look at more advanced methods of optimization and how can we use them to train deep networks. In this book, we divide optimization algorithms into two camps:

- First-order
- Second-order

First-order optimization algorithms calculate the *Jacobian* matrix.

The Jacobian

The Jacobian is a matrix of partial derivatives of loss function values with respect to each parameter.

The Jacobian has one partial derivative per parameter (to calculate partial derivatives, all other variables are momentarily treated as constants). The algorithm then takes one step in the direction specified by the Jacobian.

Second-order algorithms calculate the derivative of the Jacobian (i.e., the derivative of a matrix of derivatives) by approximating the *Hessian*. Second-order methods take into account interdependencies between parameters when choosing how much to modify each parameter.

Second-Order Methods

Second-order methods can take "better" steps; however, each step will take longer to calculate.

Practical Use of Optimization Algorithms

We provide much of the details on optimization algorithms so that you are aware of the mechanics involved for reference. In later chapters in this book, we simplify the discussion around optimization algorithms to give you a rule of thumb for when to use which algorithm and in what context.

Other Optimization Algorithms

There are other variations of optimization algorithms (such as "meta heuristics") that we won't cover in this book. They include the following:

- Genetic algorithms
- Particle swarm optimization
- Ant colony optimization
- Simulated annealing

First-order methods

The Jacobian, as mentioned, is a matrix of partial derivatives of the loss function with respect to the parameters in the network. In practice, we calculate it at a specific point —the current values of the parameters.

If we think about taking one step at a time to reach an objective, first-order methods calculate a gradient (Jacobian) at each step to determine which direction to go in next. This means that at each iteration, or step, we are trying to find the next best possible direction to go, as defined by our objective function. This is why we consider optimization algorithms to be a "search." They are finding a path toward minimal error.

Gradient descent is a member of this path-finding class of algorithms. Variations of gradient descent exist, but at its core, it finds the next step in the right direction with respect to an objective at each iteration. Those steps move us toward a global minimum error or maximum likelihood.

Stochastic gradient descent (SGD) is machine learning's workhorse optimization algorithm. SGD trains several orders of magnitude faster than methods such as batch gradient decent, with no loss of model accuracy.

Why Is SGD Considered "Stochastic"?

This is due to how we calculate the gradient for a single input training example (or mini-batch of training examples). The computed gradient is a "noisy" approximation of the true gradient yet allows SGD to converge faster.

The strengths of SGD are easy implementation and the quick processing of large datasets. You can adjust SGD by adapting the learning rate (e.g., with methods such as Adagrad, discussed shortly) or using second-order information (i.e., the Hessian), as we'll see next. SGD is also a popular algorithm for training neural networks due to its robustness in the face of noisy updates. That is, it helps you build models that generalize well.

Other Factors in Learning Rate Adjustment

It's relevant to note that other techniques such as momentum and RMSProp can affect learning rates.

Second-order methods

All second-order methods calculate or approximate the Hessian. As described earlier, we can think of the Hessian as the derivative of the Jacobian. That is, it is a matrix of second-order partial derivatives, analogous to "tracking acceleration rather than speed." The Hessian's job is to describe the curvature of each point of the Jacobian. Second-order methods include:

- Limited-memory BFGS (L-BFGS)[9]
- Conjugate gradient[10]
- Hessian-free[11]

Think of these optimization algorithms as a black-box search algorithm that determines the best way to minimize error, given an objective and a defined gradient relative to each layer.

Making Trade-offs in Optimization

A major difference in first- and second-order methods is that second-order methods converge in fewer steps yet take more computation per step.

L-BFGS. L-BFGS is an optimization algorithm and a so-called quasi-Newton method. As its name indicates, it's a variation of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, and it limits how much gradient is stored in memory. By this, we mean the algorithm does not compute the full Hessian matrix, which is more computationally expensive.

L-BFGS approximates the inverse Hessian matrix to direct weight adjustments search toward more promising areas of parameter space. Whereas BFGS stores the gradient's full $n \times n$ inverse matrix, Hessian L-BFGS stores only a few vectors that represent a local approximation of it. L-BFGS performs faster because it uses approximated second-order information. L-BFGS and conjugate gradient in practice can be faster and more stable than SGD methods.

L-BFGS in Practice

Although L-BFGS has some interesting properties, they are not commonly used in practice for deep networks.

Conjugate gradient. Conjugate gradient guides the direction of the line search process based on conjugacy information. Conjugate gradient methods focus on minimizing the conjugate L2 norm. Conjugate gradient is very similar to gradient descent in that it performs line search. The major difference is that conjugate gradient requires each

9 Le et al. 2011. "On Optimization Methods for Deep Learning."

10 LeCun et al. 1998. "Efficient BackProp."

11 Martens. 2010. "Deep learning via Hessian-free optimization."

successive step in the line search process to be conjugate to one another with respect to direction.

Hessian-free. Hessian-free optimization is related to Newton's method, but it better minimizes the quadratic function we get. It is a powerful optimization method adapted to neural networks by James Martens in 2010. We find the minimum of the quadratic function with an iterative method called conjugate gradient.

Hyperparameters

Here we define a hyperparameter as any configuration setting that is free to be chosen by the user that might affect performance.

Hyperparameters fall into several categories:

- Layer size
- Magnitude (momentum, learning rate)
- Regularization (dropout, drop connect, L1, L2)
- Activations (and activation function families)
- Weight initialization strategy
- Loss functions
- Settings for epochs during training (mini-batch size)
- Normalization scheme for input data (vectorization)

In this section, we expand on the concepts from Chapter 1 with some new hyperparameters relevant to deep learning training.

A Few Cautionary Notes About Hyperparameters

Some hyperparameters apply only some of the time. We cover the details of this more in Chapters 6 and 7. Moreover, changing a specific hyperparameter might affect the best settings for other hyperparameters. We'd also like to point out that some hyperparameters are incompatible with one another (e.g., Adagrad + momentum).

Layer size

Layer size is defined by the number of neurons in a given layer. Input and output layers are relatively easy to figure out because they correspond directly to how our modeling problem handles input and ouput. For the input layer, this will match up to the number of features in the input vector. For the output layer, this will either be a single output neuron or a number of neurons matching the number of classes we are trying to predict.

Deciding on neuron counts for each hidden layer is where hyperparameter tuning becomes a challenge. We can use an arbitrary number of neurons to define a layer

and there are no rules about how big or small this number can be. However, how complex of a problem we can model is directly correlated to how many neurons are in the hidden layers of our networks. This might push you to begin with a large number of neurons from the start but these neurons come with a cost.

Depending on the deep network architecture, the connection schema between layers can vary. However, the weights on the connections, as we saw in Chapter 1, are the parameters we must train. As we include more parameters in our model, we increase the amount of effort needed to train the network. Large parameter counts can lead to long training times and models that struggle to find convergence.

Large Parameter Counts and Overfitting

There are also cases in which a larger model will sometimes converge easier because it will simply "memorize" the training data. In Chapter 6, we further discuss ways to deal with overfitting.

In Chapter 6, we look at heuristics for determining neuron count per layer and how to iteratively find good layer hyperparameters.

Magnitude hyperparameters

Hyperparameters in the magnitude group involve the gradient, step size, and momentum.

Learning rate. The learning rate in machine learning is how fast we change the parameter vector as we move through search space. If the learning rate becomes too high, we can move toward our goal faster (least amount of error for the function being evaluated), but we might also take a step so large that we shoot right past the best answer to the problem, as well.

High Learning Rates and Stability

Another side effect of learning rates that are large is that we run the risk of having unstable training that does not converge over time.

If we make our learning rate too small, it might take a lot longer than we'd like for our training process to complete. A low learning rate can make our learning algorithm inefficient. Learning rates are tricky because they end up being specific to the dataset and even to other hyperparameters. This creates a lot of overhead for finding the right setting for hyperparameters.

We also can schedule learning rates to decrease over time according to some rule. We'll learn more about this in Chapters 6 and 7.

The Importance of Learning Rate as a Hyperparameter

Learning rate is considered one of the key hyperparameters in neural networks.

Nesterov's momentum. The "vanilla" version of SGD uses gradient directly, and this can be problematic because gradient can be nearly zero for any parameter. This causes SGD to take tiny steps in some cases, and steps that are too big for situations in which the gradient is too large. To alleviate these issues, we can use techniques such as the following:

- Nesterov's momentum
- RMSProp
- Adam
- AdaDelta

DL4J and Updaters

Nesterov's momentum, RMSProp, Adam, and AdaDelta are known as "updaters" in the terminology of DL4J. Most of the terms used in this book are universal for most of all deep learning literature; we just wanted to note this variation for DL4J specifically.

We can speed up our training by increasing momentum, but we might lower the chance that the model will reach minimal error by overshooting the optimal parameter values. Momentum is a factor between 0.0 and 1.0 that is applied to the change rate of the weights over time. Typically, we see the value for momentum between 0.9 and 0.99.

AdaGrad. AdaGrad[12] is one technique that has been developed to help augment finding the "right" learning rate. AdaGrad is named in reference to how it "adaptively" uses subgradient methods to dynamically control the learning rate of an optimization algorithm. AdaGrad is monotonically decreasing and never increases the learning rate above whatever the base learning rate was set at initially.

AdaGrad is the square root of the sum of squares of the history of gradient computations. AdaGrad speeds our training in the beginning and slows it appropriately toward convergence, allowing for a smoother training process.

12 Duchi, Hazan, and Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization."

RMSProp. RMSprop is a very effective, but currently unpublished adaptive learning rate method. Amusingly, everyone who uses this method in their work currently cites slide 29 of Lecture 6 of Geoff Hinton's Coursera class.

AdaDelta. AdaDelta[13] is a variant of AdaGrad that keeps only the most recent history rather than accumulating it like AdaGrad does.

ADAM. ADAM (a more recently developed updating technique from the University of Toronto) derives learning rates from estimates of first and second moments of the gradients.

Regularization

Let's dig deeper into the idea of regularization that we touched on in Chapter 2. Regularization is a measure taken against overfitting. Overfitting occurs when a model describes the training set but cannot generalize well over new inputs. Overfitted models have no predictive capacity for data that they haven't seen. Geoffery Hinton described the best way to build a neural network model:

> Cause it to overfit, and then regularize it to death.

Regularization for hyperparameters helps modify the gradient so that it doesn't step in directions that lead it to overfit. Regularization includes the following:

- Dropout
- DropConnect
- L1 penalty
- L2 penalty

Dropout and DropConnect mute parts of the input to each layer, such that the neural network learns other portions. Zeroing-out parts of the data causes a neural network to learn more general representations. Regularization works by adding an extra term to the normal gradient computed.

Dropout. Dropout[14] is a mechanism used to improve the training of neural networks by omitting a hidden unit. It also speeds training. Dropout is driven by randomly dropping a neuron so that it will not contribute to the forward pass and backpropagation.

13 Zeiler. 2012. "ADADELTA: An Adaptive Learning Rate Method."

14 Bengio et al. 2015. *Deep Learning* (In Preparation).

Dropout Related to Model Averaging

We also can relate dropout to the concept of averaging the output of multiple models. If we use a dropout coefficient of 0.5, we have the mean of the model. A random Dropout of features is a sampling from 2^N possible architectures, where N is the number of parameters.

DropConnect. DropConnect[15] does the same thing as Dropout, but instead of choosing a hidden unit, it mutes the connection between two neurons.

L1. The penalty methods L1 and L2, in contrast, are a way of preventing the neural network parameter space from getting too big in one direction. They make large weights smaller.

L1 regularization is considered computationally inefficient in the nonsparse case, has sparse outputs, and includes built-in feature selection. L1 regularization multiplies the absolute value of weights rather than their squares. This function drives many weights to zero while allowing a few to grow large, making it easier to interpret the weights.

L2. In contrast, L2 regularization is computationally efficient due to it having analytical solutions and nonsparse outputs, but it does not do feature selection automatically for us. The "L2" regularization function, a common and simple hyperparameter, adds a term to the objective function that decreases the squared weights. You multiply half the sum of the squared weights by a coefficient called the weight-cost. L2 improves generalization, smooths the output of the model as input changes, and helps the network ignore weights it does not use.

Mini-batching

With mini-batching,[16] we send more than one input vector (a group or batch of vectors) to be trained in the learning system. This allows us to use hardware and resources more efficiently at the computer-architecture level. This method also allows us to compute certain linear algebra operations (specifically matrix-to-matrix multiplications) in a vectorized fashion. In this scenario we also have the option of sending the vectorized computations to GPUs if they are present.

15 *Ibid.*

16 Bengio. 2012. "Practical recommendations for gradient-based training of deep architectures."

Summary

In Chapter 2, we learned about some of the basic regularization tools for feed-forward multilayer neural networks. In this chapter, we expanded this definition to some newer techniques and options for hyperparameters to find better parameter vectors. Let's now put some of these ideas together to construct building blocks for deep networks.

Building Blocks of Deep Networks

Building deep networks goes beyond basic feed-forward multilayer neural networks. In some cases, deep networks combine smaller networks as building blocks into larger networks; in other cases, they use a specialized set of layers. Here are the specific building blocks we want to highlight:

- Feed-forward multilayer neural networks
- RBMs
- Autoencoders

In Chapter 2, we introduced the canonical feed-forward networks. Inspired by networks of biological neurons, feed-forward networks are the simplest artificial neural networks. They are composed of an input layer, one or many hidden layers, and an output layer. In this section, we introduce networks that are considered building blocks of larger deep networks:

- RBMs
- Autoencoders

Both RBMs and autoencoders are characterized by an extra layer-wise step for training. They are often used for the pretraining phase in other larger deep networks.

Unsupervised Layer-Wise Pretraining

Unsupervised layer-wise pretraining[17] can help in some training circumstances. Over time, better optimization methods, activation functions, and weight initialization methods have lessened the importance of pretraining-based deep networks. A case for which pretraining becomes interesting is when we have a lot of unlabeled data yet only a relatively smaller set of labeled training data. Pretraining does, however, add extra overhead regarding tuning and extra training time.

Layer-wise pretraining works by performing unsupervised pretraining of the first layer (e.g., RBMs) based on the input data. This gives us the first layer of weights for our main neural network (e.g., feed-forward multilayer perceptron). We perform this process for each layer progressively in the network, using the output of previous layers based on the training input as the input to the successive layers. This pretraining process allows us to initialize the main neural network's parameters with good initial values.

RBMs model probability and are great at feature extraction. They are feed-forward networks in which data is fed through them in one direction with two biases rather than one bias as in traditional backpropagation feed-forward networks.

Autoencoders are a variant of feed-forward neural networks that have an extra bias for calculating the error of reconstructing the original input. After training, autoencoders are then used as a normal feed-forward neural network for activations. This is an unsupervised form of feature extraction because the neural network uses only the original input for learning weights rather than backpropagation, which has labels. Deep networks can use either RBMs or autoencoders as building blocks for larger networks (it is, however, rare that a single network would use both). In the following sections, we take a closer look at both networks.

RBMs

RBMs are used in deep learning for the following:

- Feature extraction
- Dimensionality reduction

The "restricted" part of the name "Restricted Boltzmann Machines" means that connections between nodes of the same layer are prohibited (e.g., there are no visible-visible or hidden-hidden connections along which signal passes). Geoff Hinton, the

17 Bengio et al. 2007. "Greedy Layer-Wise Training of Deep Networks."

deep learning pioneer who popularized RBM use almost a decade ago, describes the more general Boltzmann machine as follows:

> A network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off.

RBMs are also a type of autoencoder, which we'll talk about in a following section. RBMs are used for pretraining layers in larger networks such as Deep Belief Networks.

Geoff Hinton

Geoff Hinton is a distinguished researcher at Google (part-time) and also works for the University of Toronto as a Distinguished Emeritus Professor. Dr. Hinton's team is credited for key work on RBMs and Deep Belief Networks.

Dr. Hinton's team has produced results that in no small part helped drive the wide-scale interest in the field of deep learning we see today. In 2012, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton developed a "large, deep convolutional neural network" that won the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) known as "AlexNet." AlexNet[18] was hailed as an advancement in computer vision, and some point to it specifically as the beginning of the deep learning craze.

Dr. Hinton has been a proponent of basic research and the long-term persistence it takes to get results:

> ...it took 17 years after Terry Sejnowski and I invented the Boltzmann machine learning algorithm before I found a version of it that worked efficiently. If you really believe in an idea, you just have to keep trying.

Network layout

There are five main parts of a basic RBM:

- Visible units
- Hidden units
- Weights
- Visible bias units
- Hidden bias units

A standard RBM has a visible layer and a hidden layer, as shown in Figure 2-3. We can also see a graph of weights (connections) between the hidden and visible units in

18 Krizhevsky, Sutskever, and Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks."

the figure. Think of these weights in the same way you think of weights in the classical neural network sense.

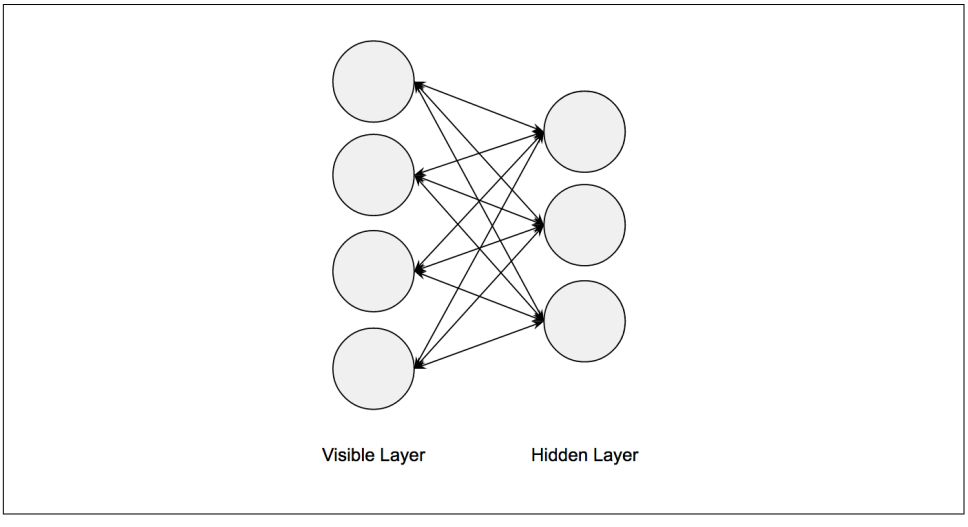

Figure 2-3. RBM network

With RBMs, every visible unit is connected to every hidden unit, yet no units from the same layer are connected. Each layer of an RBM can be imagined as a row of nodes. The nodes of the visible and hidden layers are connected by connections with associated weights.

Visible and hidden layers. In an RBM, every single node of the input (visible) layer is connected by weights to every single node of the hidden layer, but no two nodes of the same layer are connected. The second layer is known as the "hidden" layer. Hidden units are *feature detectors*, learning features from the input data. Nodes in each layer are biologically inspired as with the feed-forward multilayer neural network from Chapter 1. Units (nodes) in the visible layer are "observable" in that they take training vectors as input. Each layer has a bias unit with its state always set to on.

Each node performs computation based on the input to the node and outputs a result based on a stochastic decision whether or not to transmit data through an activation. Just as with the artificial neuron from Chapter 1, the activation computation is based on weights on the connections and the input values. The initial weights are randomly generated.

Connections and weights. All connections are visible-hidden; none are visible-visible or hidden-hidden. The edges represent connections along which signals are passed. Loosely speaking, those circles, or nodes, act like human neurons. They are decision units. They make decisions about whether to be on or off through acts of computa-

tion. "On" means that they pass a signal further through the net; "off" means that they don't.

Usually, being "on" means the data passing through the node is valuable; it contains information that will help the network make a decision. Being "off" means the network thinks that particular input is irrelevant noise. A network comes to know which features/signals correlate with which labels (which code contains which messages) by being trained. With training, networks learn to make accurate classifications of the input they receive.

Biases. There is a set of bias weights ("parameters") connecting the bias unit for each layer to every unit in the layer. Bias nodes help the network better triage and model cases in which an input node is always on or always off.

Training

The technique known as *pretraining* using RBMs means teaching it to reconstruct the original data from a limited sample of that data. That is, given a chin, a trained network could approximate (or "reconstruct") a face. RBMs learn to reconstruct the input dataset. We'll review the concept of reconstruction in the next section.

Contrastive Divergence

RBMs calculate gradients by using an algorithm called contrastive divergence. Contrastive divergence is the name of the algorithm used in sampling for the layer-wise pretraining of a RBM. Also called CD-k, contrastive divergence minimizes the KL divergence (the delta between the real distribution of the data and the guess) by sampling *k* steps of a Markov chain to compute a guess.

Reconstruction

Deep neural networks with unsupervised pretraining (RBMs, autoencoders) perform feature engineering from unlabeled data through reconstruction. In pretraining, the weights learned through unsupervised pretrain learning are used for weight initialization in networks such as Deep Belief Networks.

Reconstruction as Matrix Factorization

Reconstruction is a matrix factorization problem (also known as matrix decomposition).

Figure 2-4 presents a visual explanation of the network involved in reconstruction in RBMs.

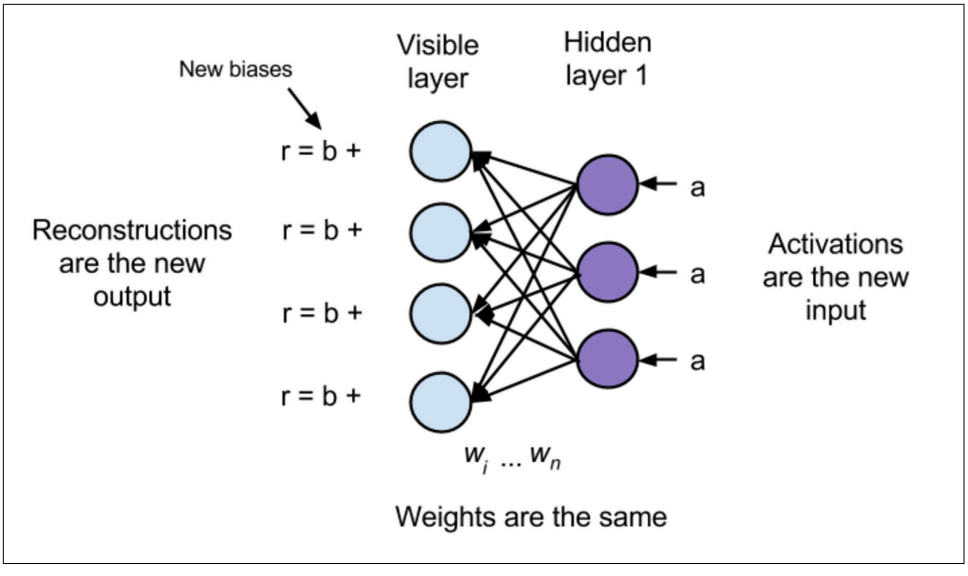

Figure 2-4. Reconstruction in RBMs

We can visually explain reconstruction in RBMs by looking at the MNIST dataset. MNIST stands for the *mixed National Institute of Standards and Technology* dataset that contains the images. The MNIST dataset is a collection of images representing the handwritten numerals 0 through 9. Figure 2-5 depicts a sample of some of the handwritten digits in MNIST.

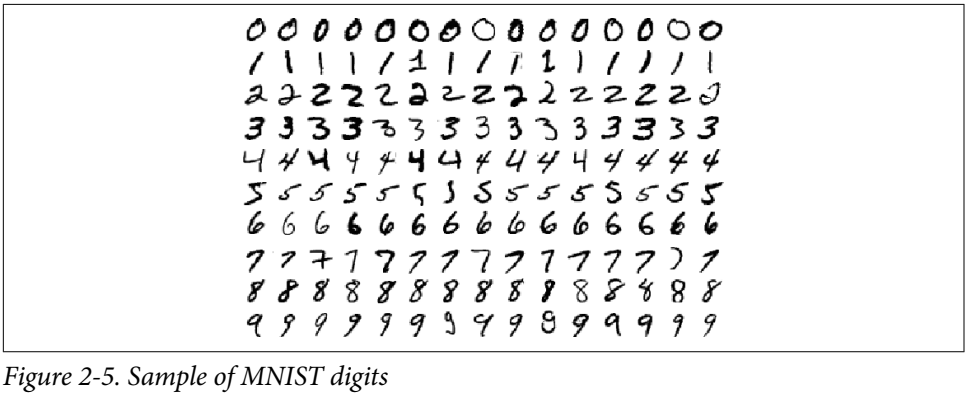

Figure 2-5. Sample of MNIST digits

The training dataset in MNIST has 60,000 records and the test dataset has 10,000 records. If we use a RBM to learn the MNIST dataset, we can sample from the trained

network to see[19] how well it can reconstruct the digits. Figure 2-6 shows renderings of MNIST digits being progressively reconstructed with a RBM.

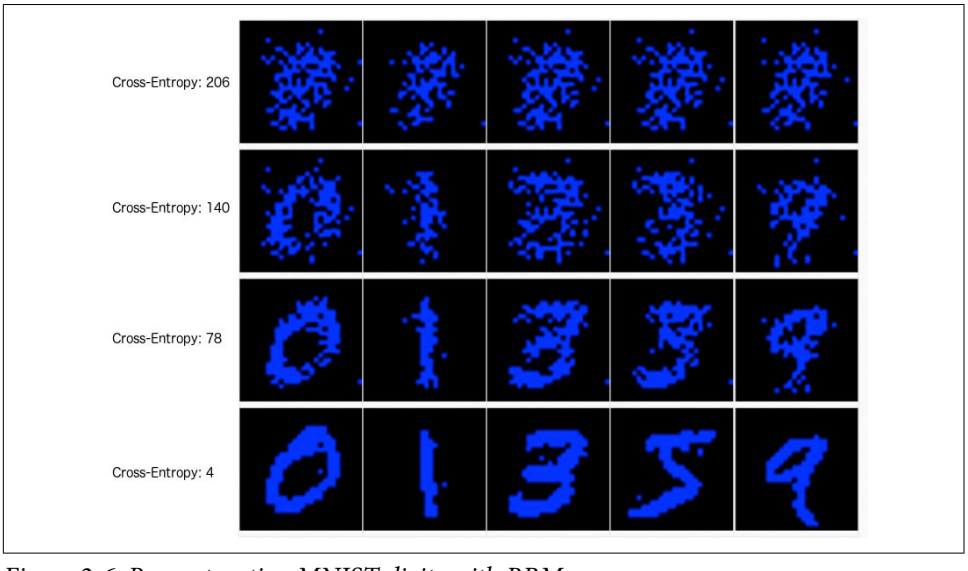

Figure 2-6. Reconstructing MNIST digits with RBMs

If the training data has a normal distribution, most of them cluster around a central *mean*, or average, and become scarcer the further you stray from that average. It looks like a bell curve. If we know the mean and the variance, or sigma, of normal data, we can reconstruct that curve. But suppose that we don't know the mean and variance. Those are parameters we then need to guess. Picking them randomly and contrasting the curve they produce with the original data can operate similarly to a loss function. We measure the difference between two probability distributions much like we measure erroneous classifications, adjust our parameters, and try again.

Reconstruction Cross-Entropy

The objective function here is usually reconstruction cross-entropy, or KL divergence (the mathematicians and cryptanalysts Solomon Kullback and Richard Leibler first published a paper on the technique in 1951). "Cross" refers to the comparison between two distributions. "Entropy" is a term from information theory that refers to uncertainty. For example, a normal curve with a wide spread, or variance, also implies more uncertainty about where data points will fall. That uncertainty is called entropy.

19 Yosinksi and Lipson. 2012. "Visually Debugging Restricted Boltzmann Machine Training with a 3D Example."

Other uses of RBMs

Here are some other places we see RBMs used:

- Dimensionality reduction
- Classification
- Regression
- Collaborative filtering
- Topic modeling

Autoencoders

We use autoencoders to learn compressed representations of datasets. Typically, we use them to reduce a dataset's dimensionality. The output of the autoencoder network is a reconstruction of the input data in the most efficient form.

Similarities to multilayer perceptrons

Autoencoders share a strong resemblance with multilayer perceptron neural networks in that they have an input layer, hidden layers of neurons, and then an output layer. The key difference to note between a multilayer perceptron network diagram (from earlier chapters) and an autoencoder diagram is the output layer in an autoencoder has the same number of units as the input layer does.

Figure 2-7 presents an example of an autoencoder network.

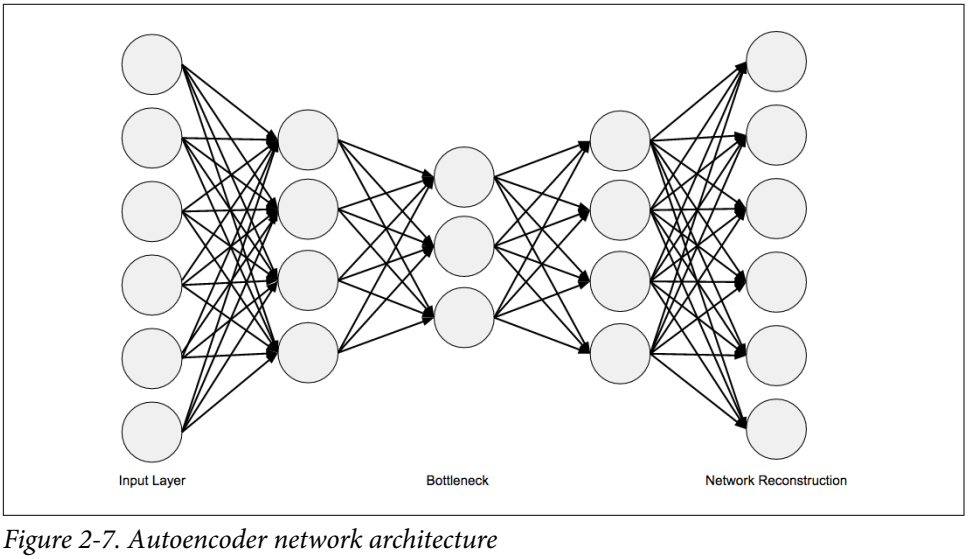

Figure 2-7. Autoencoder network architecture

Beyond the output layer, there a few other differences, which we outline in the next section.

Defining features of autoencoders

Autoencoders differ from multilayer perceptrons in a couple of ways:

- They use unlabeled data in unsupervised learning.
- They build a compressed representation of the input data.

Unsupervised learning of unlabeled data. The autoencoder learns directly from unlabeled data. This is connected to the second major difference between multilayer perceptrons and autoencoders.

Learning to reproduce the input data. The goal of a multilayer perceptron network is to generate predictions over a class (e.g., fraud versus not fraud). An autoencoder is trained to reproduce its own input data.

Training autoencoders

Autoencoders rely on backpropagation to update their weights. The main difference between RBMs and the more general class of autoencoders is in how they calculate the gradients.

Common variants of autoencoders

Two important variants of autoencoders to note are *compression autoencoders* and *denoising autoencoders*.

Compression autoencoders. This is the architecture depicted in Figure 2-7. The network input must pass through a bottleneck region of the network before being expanded back into the output representation.

Denoising autoencoders. The denoising autoencoder[20] is the scenario in which the autoencoder is given a corrupted version (e.g., some features are removed randomly) of the input and the network is forced to learn the uncorrupted output.

Applications of autoencoders

Building a model to represent the input dataset might not sound useful on the surface. However, we're less interested in the output itself and more interested in the difference between the input and output representations. If we can train a neural

20 Vincent et al. 2010. "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion."

network to learn data it commonly "sees," then this network can also let us know when it's "seeing" data that is unusual, or anomalous.

Autoencoders as Anomaly Detectors

Autoencoders are commonly used in systems in which we know what the normal data will look like, yet it's difficult to describe what is anomalous. Autoencoders are good at powering anomaly detection systems.

Variational Autoencoders

A more recent type of autoencoder model is the variational autoencoder (VAE) introduced by Kingma and Welling[21] (see Figure 2-8). The VAE is similar to compression and denoising autoencoders in that they are all trained in an unsupervised manner to reconstruct inputs.

However, the mechanisms that the VAEs use to perform training are quite different. In a compression/denoising autoencoder, activations are mapped to activations throughout the layers, as in a standard neural network; comparatively, a VAE uses a probabilistic approach for the forward pass.

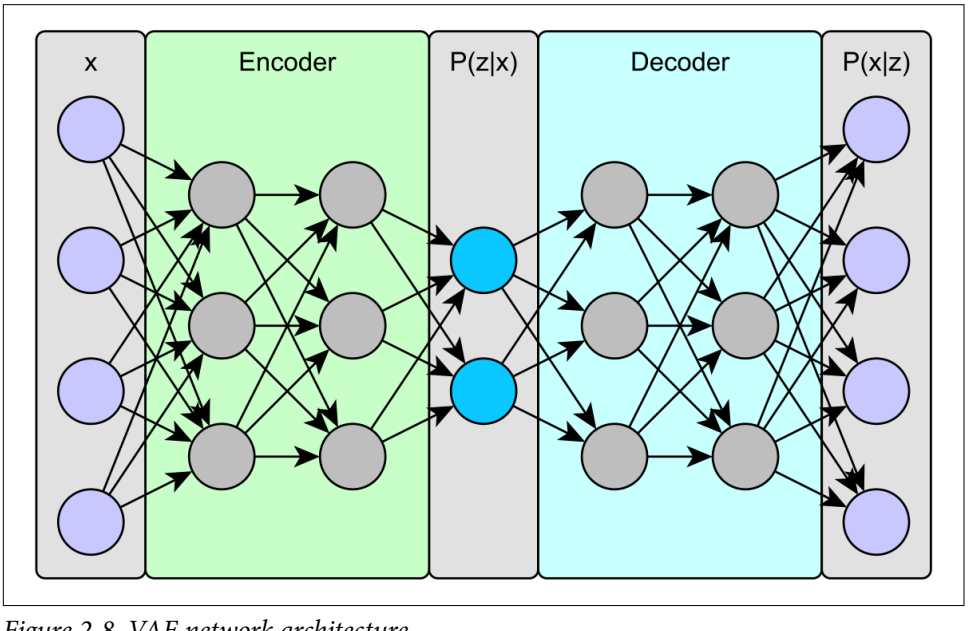

Figure 2-8. VAE network architecture

21 Kingma and Welling. 2013. "Auto-Encoding Variational Bayes."

The VAE model assumes that the data x is generated in two steps: (a) a value $z^i \sim p(z)$ is generated from a prior distribution, and (b) the data instance is generated according to some conditional distribution $x^i \sim p(x|z)$. Of course, we don't actually know the values of z, and inferring $p(z|x)$ exactly is generally intractable. To handle this, we let both distributions, $p(z|x)$ and $p(x|z)$, be approximated by neural networks—the encoder and decoder, respectively. For example, if the $P(z|x)$ is Gaussian, the encoder forward-pass activations provide the Gaussian distribution parameters μ and σ^2.

Similarly, the distribution parameters for $P(x|z)$ are provided by the decoder forward pass.[22] Overall, the network is trained by backpropagation to maximize a lower bound on the marginal likelihood of the training data, $\log p\left(x^1, ..., x^N\right)$. The VAE model also has been extended to allow unsupervised learning on time series with the variational *recurrent* autoencoder.[23] In Chapter 5, we see the VAE in a practical example generating MNIST digits.

22 These distribution parameters should not be confused with the trainable network parameters: in practice, they are just network activations used to specify (for example) the mean and variance values for a Gaussian distribution, or the mean value for Bernoulli distribution.

23 Fabius and van Amersfoort. 2014. "Variational Recurrent Auto-Encoders."

About the Authors

Josh Patterson currently is the head of field engineering at Skymind. Josh previously ran a consultancy in the big data/machine learning/deep learning space. Previously, he worked as a principal solutions architect at Cloudera and as a machine learning/distributed systems engineer at the Tennessee Valley Authority, where he brought Hadoop into the smart grid with the openPDC project. Josh has a Master's in computer science from the University of Tennessee at Chattanooga where he published research on mesh networks (tinyOS) and social insect optimization algorithms. Josh has more than 17 years in software development and is very active in the open source space, contributing to projects such as DL4J, Apache Mahout, Metronome, IterativeReduce, openPDC, and JMotif.

Adam Gibson is a deep learning specialist based in San Francisco. He works with *Fortune 500* companies, hedge funds, PR firms, and startup accelerators to create their machine learning projects. Adam has a strong track record helping companies handle and interpret big realtime data. He has been a computer nerd since the age of 13, and actively contributes to the open source community through *http://deeplearning4j.org*.

Colophon

The animal on the cover of *Deep Learning* is the oarfish (*Regalecus glesne*), a large lampriform (ray-finned) fish native to temperate and tropical oceans. They have a long and slender body with spiny dorsal fins running down their back. Oarfish can grow up to 11 meters in length, making them the largest bony fish in the world.

Oarfish are solitary animals and are rarely seen by humans. They spend much of their time in the mesopelagic zone (200 to 1,000 meters deep) and only go to the surface when they are sick or injured. Oarfish are carnivores that feed primarily on zooplankton as well as small fish, jellyfish, and squid.

The meat of the oarfish has a gelatinous consistency, so it is not targeted by commercial fishermen. Humans generally encounter the species only when dead or dying oarfish wash up on shore. Because of their size and shape, it is thought that oarfish may be the basis for sea serpent legends. Although their total population is unknown, there are no known environmental threats to oarfish.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Braukhaus Lexicon*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.