Chapter 18

Machine Learning

We now live in the age of machine learning, the technology that powers spam filters, image classifiers, automated translators, smart assistants (like Siri and Alexa), self-driving cars, and a multitude of other recently-released or soon-to-be-released products. This is only the beginning. The techniques behind contemporary machine learning are still in their infancy, and we have only just begun to understand how to use them to their full potential.

Perhaps to your surprise, the most popular tools for performing machine learning are written in Python. In this chapter, we will teach you basic proficiency with two libraries, `scikit-learn` and `keras`, that are used to develop industrial-strength machine learning models. Although machine learning has its own mystique above and beyond the mere act of programming, it is entirely within the reach of any reader who has made it to this point in the book.

In this chapter, we will discuss the conceptual principles of machine learning (Section 18.1) before concretely presenting two machine learning techniques in particular: logistic regression (Section 18.2) and neural networks (Section 18.3).

Logistic regression is a standard statistical technique that allows you to predict whether a piece of data is a member of one of two categories. For example, one could use logistic regression to predict whether a picture contains a cat or a dog or whether a Supreme Court justice is likely to vote for or against one side of a case.

Neural networks, whose name suggests yet further mystique, are really just a further refinement of logistic regression. (They have nothing to do with the human brain, regardless of what the name may suggest.) However, this refinement gives neural networks dramatically more expressive power. Neural networks power nearly all modern machine learning applications: personal assistants, automated translation, image classification, and self-driving cars. Not only is the topic of neural networks vast, but it is evolving quickly. In Section 18.3, we cover fundamentals that have withstood the test of time, giving you a jumping off point for further learning.

Entire books have been written on machine learning, and we cannot convey all of that knowledge in a single chapter. Instead, our goal is to give you enough information so that you are capable of reasoning fluently about machine learning, making basic use of two specific techniques, and continuing to learn on your own.

Reassurance for the mathematically inhibited. As you are no doubt aware by this point in this book, programming does involve a small amount of elementary math. Although we have labored to keep these mathematical intrusions to a minimum, numbers have surfaced from time to time. As we acknowledged at the beginning of this book, lawyers tend to have an unfounded, yet pathological, aversion to math. If you have made it this far, we congratulate you for having overcome those inhibitions.

We repeat and emphasize the same advice here. Machine learning is based on statistical machinery that will require you to do some amount of interacting with numbers. The mathematical concepts necessary for this chapter are slightly more advanced; however, we assure you that you learned all of the tools you need for this chapter in high school, and we will refresh your memory in the few moments where doing so is necessary. Underpinning both logistic regression and neural networks is a small amount of calculus, but the libraries we use hide this away completely. Just as you have done throughout this book so far, we ask that you trust us to guide you through this math unharmed.

18.1 Principles of Machine Learning

What, exactly, is machine learning? We define machine learning using two criteria:

1. Machine learning involves developing a model—a simplified version—of some process (e.g., predicting how a Supreme Court justice will vote on a particular case). This model should have predictive power (i.e., it should be at least somewhat accurate).

2. This model is developed by learning from existing data (e.g., how a justice has voted in previous cases) in order to make predictions about new data (e.g., how a justice will vote in the future).

In this section, we will sharpen our understanding of both of these qualities and discuss the way engineers set up and solve machine learning problems. At its heart, machine learning is essentially automated pattern recognition. It aims to extract a pattern from existing data that is useful for making sense of new data.

Where does the term artificial intelligence (which we have carefully avoided to this point) fit into this picture? Artificial intelligence is a vast, amorphous, and ambiguous term that captures any computer-aided reasoning. Machine learning is certainly a kind of artificial intelligence. The challenge with the term "artificial intelligence" is that, depending on how it is defined, any computer program can fall within its purview. Does the program we wrote assessing whether a user is eligible for a public benefit program qualify as artificial intelligence? It's hard to say. Does it qualify as machine learning? Not according to our definition: there is no process of learning from data to develop a model.

Machine learning is currently the dominant way of achieving artificial intelligence, and it provides a well-developed framework for situating problems that we hope to solve with artificial intelligence. All of the concepts and techniques in ths chapter are specific to machine learning.

18.1.1 Models

A model is a simplified (often oversimplified) way of summarizing a much more complicated process. Models typically aim to make predictions about how changes in circumstances will lead to changes in outcomes. For example, an economic model might aim to predict how a tax cut (i.e., a change in economic circumstances) will affect economic growth, unemployment, and future tax revenues for the government (i.e., changes in outcomes).

By definition, models are not correct. By simplifying the world, a model will inherently sacrifice some degree of accuracy. However, it is often productive to sacrifice a small amount of accuracy to achieve a vastly simpler model. As a famous statement by statistician George Box goes, "All models are wrong, but some models are useful."

Types of machine learning tasks. Most machine learning tasks can be grouped into two categories: classification and regression. Classification tasks involve predicting whether a particular input falls into one of several categories. For example, we might wish to predict whether a particular Supreme Court case will be decided in favor of the petitioner (a category) or in favor of the respondent (another category). Likewise, one might wish to determine whether an FCC comment was favorable to a rule (a category) or unfavorable (another category). Finally, we might wish to analyze writing style to determine which of nine possible Supreme Court justices (nine different categories) wrote a particular opinion. In practice, classification models typically predict the probability (i.e., the percent chance) that an input falls into a particular category rather than making a clear decision to pick one category over the others. We will explore this idea later in the chapter.

In regression, we try to predict a specific output for each input. For example, a model might aim to predict the change in GDP attributable to a change in tax policy. The model predicts a concrete number, for example that GDP will increase by 0.3%. As another example, a model might try to predict the number of representatives who will vote for a particular bill.

In this chapter, we will mainly consider models that perform classification rather than regression, but it is important to keep in mind that regression is an equally natural form of machine learning model.

Parameterizable models. In most machine learning contexts, we usually consider models that are parameterizable. To understand this concept, consider an example: we want to develop a model that predicts which side a Supreme Court justice will vote on a particular case based on the number of questions she asks both sides. Initially, our model might predict that she will vote against the side to whom she asked more questions.

However, we may have to change this model. Perhaps justices are reluctant to reverse the decisions of lower courts. It might be that a better model predicts that the justice will give some degree of deference to the side that won in the lower court even if she asks it slightly more questions. Concretely, one possible model is that the justice will vote for the side that won in the lower court unless she asked that side, say, five more questions than she asked the side that lost in the lower court. In other words, asking five more questions of the side that won in lower court overcomes any deference the justice might previously have granted. The number five is arbitrary. The correct number might be two, seven, twelve, or even zero (i.e., no deference). The number of questions is a parameter that we can change about our model depending on what we learn from looking at real data—previous Supreme Court transcripts.

In machine learning, we usually begin by selecting a model family—a model with one or more parameters that can be changed to create a wide variety of different models. In the preceding example, the model family considers the difference between the number of questions a justice asked to the two sides in a Supreme Court case. For example, the justice asked one side five questions more than the other. It has a single parameter: the threshold at which this difference overcomes the deference due to a lower court's decision.

18.1.2 Learning

The other distinguishing quality of machine learning is that the parameters for a model are selected by examining data about which we already know the answers. For example, to develop a model that predicts how a justice will vote on future cases, we consider how she voted on previous cases. The act of tuning the parameters of a model based on pre-existing data whose answers are known ahead of time is known as learning or training. The pre-existing data itself is known as training data. The act of using a model to make predictions about new data is known as inference.

The typical workflow for creating and using a machine learning model is as follows:

1. Select a particular model family that we wish to train.
2. Acquire some training data that includes both the inputs that we intend to provide to our model (e.g., transcripts of Supreme Court cases) and the actual outcomes (e.g., how a justice voted).
3. Use the training data to find the parameters that make the model as accurate as possible.
4. As necessary, supply new input data (whose answers are unknown) to the model and ask it to make predictions about what the answers will be.

It is important to note that training is typically performed in its entirety before the model is ever used to make predictions. After the model is deployed for inference, no further training ever occurs. There are occasionally some situations in which a model is intermittently trained, utilized, and further re-trained, but they are relatively rare in practice.

Data and features. How, exactly, do we turn complicated information (like the transcript of a Supreme Court oral argument) into concise data on which a model can operate (whether performing training or inference)? Before providing data to a model, we typically break it down into smaller pieces that summarize interesting aspects of the data. For example, the number of questions a justice asked to each side in a case. Something is clearly lost when simplifying data down from the raw transcript of an oral argument to the number of questions that were asked. However, the number of questions (two integers) is much easier to mathematically process than the transcript (a string containing many words spoken by many participants) and may distill important information that a model would have a hard time extracting on its own.

If we wanted to, we could provide the model with many such data points about each case, such as the amount of time that the justice spent speaking and whether each question was supportive or critical of the particular side's case. Each of these data points is known as a feature, and the process of turning raw data into a small set of features usable by a model is known as feature extraction. Feature extraction is very closely related to the process of data cleaning, in which raw data is processed into a more regular, structured form amenable to analysis.

Training, testing, and validation. Even after we have performed feature-extraction on our training data to convert each entry from raw information into usable features, we must still perform one further step before we are ready to train our model. Specifically, we randomly divide this data into three parts which we call training data, validation data, and testing data. We will train the model using the training data and evaluate how well it performs using the validation and test data.

To understand why it is essential that we perform this extra step, consider a motivating problem: once we have trained our model, how do we know how well it performs? At first glance, one idea is to see how accurately it makes the right predictions on the training data. After all, we want our model to perform as accurately as possible, so this is a reasonable way of measuring success. The problem with this approach is that our model can perform exceedingly accurately by simply memorizing the training data. If it does so, it will only work on the training data and fail to generalize to new data, meaning its performance will be terrible on new data. The act of memorizing the training data at the cost of generalization is known as overfitting. Overfitting to training data is a persistent problem in machine learning, even for experienced professionals.

In order to detect and avoid overfitting, we make use of the validation data and test data that we set aside at the beginning of this subsection. We train the model parameters on the training data and evaluate the performance of the model on the validation data. Since the model has never seen the validation data before, trying the model on the validation data gives us a way of detecting overfitting and changing our training procedure to prevent it from happening.

Unfortunately, we can sometimes accidentally change the training procedure so many times that we overfit our training procedure to the validation data. In order to detect this kind of overfitting, we assess the final accuracy of the model on the test data. Since the model does not see the test data until we are completely done developing the model, we can be assured that the test data gives us an accurate portrayal of the model's performance.

Optimization and loss. When we train a model, what is our goal? Training is a form of optimization in which we try to find the parameters that give our model the best possible performance. But how do we define the "best possible performance?" Doing so varies widely from problem to problem. Typical choices for regression and classification are completely different, and there is a wide range of typical choices. We typically try to develop a formula that characterizes how "incorrect" our model is; this formula is known as loss. Training involves finding the parameters for which our model reaches the minimum possible loss.

For classification tasks, loss is typically defined as the number of incorrect decisions that our model made on the training data. Recall that classification models don't usually output a clear decision that a particular piece of data falls into one category or the other. Instead, it will give the percent chance that it thinks a piece of data falls into each category. For example, a classifier might predict that there is a 70% chance that a justice will vote for the respondent in a particular case. A typical formula for loss calculates the difference between this percentage and the correct percentage. If the justice really did vote for the respondent, then the correct percentage was 100% and the loss is 30%. If the justice voted against the respondent, then the correct percentage was 0% and the loss is 70%.

In the sections that follow, we will discuss two specific machine learning model families and specific forumlas for loss that are popular in each of these settings.

18.2 Logistic Regression

Logistic regression is a standard statistical technique for performing classification. Given a set of training data, logistic regression learns to predict whether a set of features should be placed into one of two different categories. For example, given a set of pictures labelled as either "cat" or "dog," logistic regression could be trained to predict whether new, unlabelled images contain a cat or a dog. Later in this chapter, we will use logistic regression to build models of Supreme Court justices. Given a set of features about a case, can we predict whether a particular justice will vote for the petitioner or the respondent?

If you have taken a statistics class before, you are likely to have seen logistic regression. It is known under a variety of names (e.g., a logit model) in different fields. Logistic regression is a relatively simple machine learning technique. The underlying model family has only two parameters that are learned in the process of training a model. In contrast, neural networks often have millions of parameters. Even so, it is a very effective tool for a wide variety of classification problems.

Not only is logistic regression a valuable machine learning technique in its own right, but it is also the key building block used to create neural networks. Once you have a conceptual understanding of how logistic regression operates, neural networks are a natural next step.

In the first part of this section, we will introduce a concept that you probably learned in high school but have forgotten since: the notion of a mathematical function. The logistic function is a mathematical function that serves as our overall model family for logistic regression. In the second part of this section, we will introduce the logistic function and demonstrate how we can learn parameters for this model family to create useful machine learning models. Finally, in the third part of this section, we will use the `scikit-learn` library to build logistic regression models of Supreme Court justices in Python.

01

Unknown

Libraries to install. In this section, we will make use of several heavy-duty numerical processing and statistical libraries for Python. These libraries are designed to perform machine learning and other numerical tasks as efficiently as possible. These libraries are quite large: they will each take up tens of megabytes of space on your hard drive and therefore may take a long time to download over a slow internet collection. To follow the examples in this chapter, you will need to install:

1. `numpy`, a library for efficient numerical processing.

   ```
   $ pip3 install numpy
   ```

2. `scipy`, a library for statistical analysis.

   ```
   $ pip3 install scipy
   ```

3. `scikit-learn`, a library for machine learning (including logistic regression).

   ```
   $ pip3 install scikit-learn
   ```

Throughout this chapter, we will display several charts that visually illustrate how the logistic function behaves. All of these charts are created using Python, and we will show you the commands that were used to create these charts. If you wish to follow along with these examples, you will need to install the `matplotlib` library:

```
$ pip3 install matplotlib
```

18.2.1 Mathematical Functions

The logistic function is a mathematical function that we will use to construct machine learning models. First, we will refresh your knowledge on what, exactly, a mathematical function is. You certainly learned about it in high school, but you may not have needed to apply that concept in the time since. A mathematical function is like any other Python function. It takes as input one or more numbers and returns a single number. For example, consider the following function:

```python
def linear_function(x):
    return 2 * x
```

At this point in your Python career, this function should look rather simple. It takes as input a number, `x`, and returns twice that number, `2 * x`. For example, `linear_function(2.3)` returns `4.6` and `linear_function(-1.7)` returns `-3.4`.

As you may recall from your high school days, we can represent this function visually. To do so, we will make use of the `matplotlib` library. We do not aim to teach the `matplotlib` library in this book, but you can optionally follow the examples in this book and experiment if you'd like.

02

Unknown

```python
 1 import matplotlib.pyplot # A library for plotting graphs.
 2 import numpy # A library for numerical processing.
 3
 4 # A function.
 5 def linear_function(x):
 6     return 2 * x
 7
 8 # Create a series of inputs ranging from -10 to 10.
 9 x_values = numpy.arange(-10, 10, 0.1)
10
11 # Create the corresponding outputs of the function.
12 y_values = []
13 for x_value in x_values:
14     y_values.append(linear_function(x_value))
15
16 # Plot the values.
17 matplotlib.pyplot.plot(x_values, y_values)
18 matplotlib.pyplot.grid() # Show the grid.
19 matplotlib.pyplot.show() # Show the graph.
```

Before we show you the result of this code, we will briefly discuss what it does. The first two lines import the libraries we need to create this graph: `matplotlib.pyplot` and `numpy`. Lines 5 and 6 create the mathematical function we want to graph.

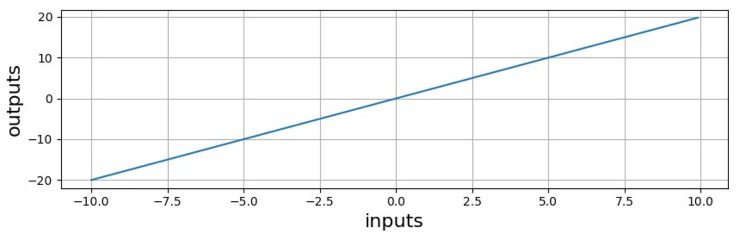
Now that we have the `linear_function` function, the rest of the program performs the steps necessary to graph it. On line 9, we create a list of inputs

03
Unknown

for this function. To do so, we use the `numpy.arange` function. This function is identical to the `range` function we have used throughout this book: it creates a list of numbers starting at the first argument (`-10`), ending just before the second argument (`10`), and increasing by the third argument at each step (`0.1`). The difference between the `range` function and the `numpy.arange` function is that the latter can operate on floating point numbers. Strangely, the built-in Python `range` function cannot. Returning to the task at hand, line 9 creates a list of floating point numbers `[-10, -9.9, -.9.8, ..., 9.8, 9.9]` and saves it to the variable `x_values`.

Lines 12 to 14 build a parallel list containing the result of running each of these numbers through `linear_function`. In other words, `y_values` will store `[-20, -19.8, -19.6, ..., 19.6, 19.8]`.

Lines 17 to 19 use `matplotlib` to create the graph below.



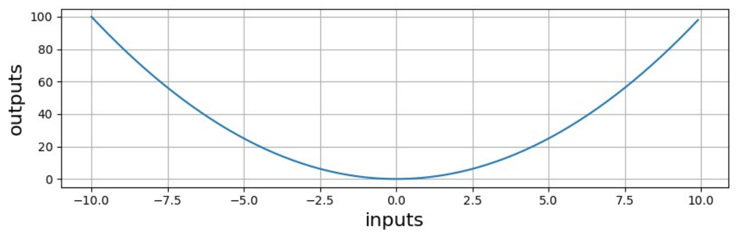
Let's take a closer look at this graph. Running left to right along the bottom of the graph are the values of the inputs that we provided to `linear_function`. Just as on line 9, the values range from -10 to 10. (As you may recall, the values running horizontally from left to right along the bottom are known as the `x-axis`.) The numbers running from bottom to top along the left of the graph are the outputs of `linear_function`. They range from `-20` to `20`, the range of outputs we produced and stored in `y_values`. (As you may recall, the values running vertically from bottom to top along the left are known as the y-axis.)

The blue line shows which input values correspond to which output values. For example, since the input value `-5` corresponds to the output value `-10`, the blue line passes through a point that is at `-5` on the x-axis (left to right) and `-10` on the y-axis (top to bottom). Likewise, since the input value `2.5` corresponds to the output value `5`, the blue line passes through a point that is at `2.5` on the x-axis and `5` on the y-axis. Since two times `0` is still `0`, the line passes through a point at `0` on the x-axis and `0` on the y-axis.

We can swap in other functions to get different graphs. For example, consider the graph produced by the function

```
def quadratic_function(x):
    return x * x
```

which squares its input (multiplies its input by itself).



The x-axis still ranges from `-10` to `10`, since those are still the inputs we provide to the function. However, the y-axis ranges from `0` to `100`. Rather than a line, we now get a curve that traces the relationship between inputs and outputs. Since `0 * 0` is still `0`, the input `0` on the x-axis corresponds to the output `0` on the y-axis. Both `5 * 5` and `-5 * -5` are `25`, so the inputs `-5` and `5` on the x-axis both correspond to the output `25` on the y-axis. Likewise, since `-10 * -10` and `10 * 10` are `100`, the inputs `-10` and `10` on the x-axis both correspond to `100` on the y-axis. The blue curve traces all of these points.

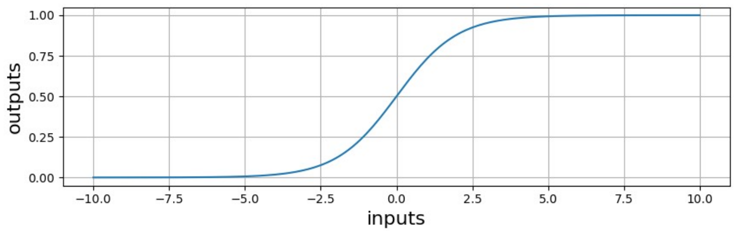
18.2.2 Logistic Regression in Concept

Now that you have refamiliarized yourself with mathematical functions and graphs, we will now introduce the logistic function, which we will use to build machine learning models. The logistic function is more complicated. You do not need to understand how this function works, and this is the first and last time in this book that you will need to think about this formula.

```
import math

def logistic_function(x):
    1.0 / (1 + math.e ** (-x))
```

To write the logistic function, we need to import the built-in Python `math` library to get access to the special value `math.e`, which is a mathematical constant like pi. Once again, you do not need to understand the math behind this function. The most important part of this function is its shape:



There are a couple of key facts about the logistic function that are useful for machine learning. First, notice that, when the input is very high, the output never gets bigger than `1`. In fact, no matter how high the input is, the output will never quite reach `1`. Likewise, no matter how low the input is, the output will never be smaller than `0`. In the middle of the graph, the output slowly switches over from being close to `0` to being close to `1` as the input grows from about `-5` to `5`. This switchover happens when the input is `0` and the output is `.5`, centered exactly between an output of `0` and an output of `1`.

How do we use this function as a machine learning model? The features that we provide to our machine learning model—a picture of an animal or data about a Supreme Court case—are the inputs to the logistic function. The output of the logistic function tells us its prediction. An output of `1` means that our model is 100% certain that the picture contains a cat or that the justice is going to vote for the petitioner. An output of `0` suggests that our model is 0% certain that the picture contains a cat or that the justice is going to vote for the petitioner, meaning, conversely, that it is 100% sure that the picture contains a dog or that the justice is going to vote for the respondent. Outputs in between `0` and `1` (e.g., `.3`) signify a certainty somewhere in between 0% and 100% (e.g., `30%`).

Consider the logistic function we have just graphed. This function represents a machine learning model that decides whether to place inputs into one of two categories which we'll abstractly call Category 0 and Category 1. In reality, these categories might be cat and dog or petitioner and respondent. Any input of `5` or greater falls into Category 1 with nearly 100% certainty and any input of `-5` or less falls into Category 0 with nearly 100% certainty since `logistic_function` outputs a clear `1` or `0`. Inputs in between are more uncertain. The input `-2.5` is 93% certain to fall into Category 0 since `logistic_function(-2.5)` is about `0.07`; similarly, the input `2.5` is 93% certain to fall into Category 1 since `logistic_function(2.5)` is about `0.93`. The model thinks that the input 0 has a 50/50 chance of being in either category, since `logistic_function(0)` is `0.5`.

Notice that this logistic regression model only works if the data actually fits exactly the right pattern. What if an input of `5` should still be in Category 0, and the model shouldn't switch over to Category 1 until the input is `10`? More concretely, consider a specific machine learning problem. Suppose we wish to model the chances that a law student will pass the bar exam based on her law school GPA. A typical GPA ranges from `0.0` to `4.0`, so the logistic regression model we just outlined doesn't make any sense. We would expect that a student with a 4.0 GPA is almost certain to pass the bar (Category 1), but a student with a 2.0 GPA may struggle to do so (Category 0). The model should cross over from predicting Category 0 to Category 1 somewhere around a 3.0 GPA.

Consider a different machine learning problem. On the weekends, Larry the lawyer sells ice cream in a local park. He wants to determine whether today will be a profitable day (Category 1) or an unprofitable day (Category 0) to sell ice cream based on the temperature. We would expect that a temperature of 80 degrees is almost certain to be a profitable day to sell ice cream and a temperature of 20 degrees is almost certain to be an unprofitable day to sell ice cream. The model should cross over from predicting Category 0 to Category 1 somewhere around 50 degrees.

In order for logistic regression to capture both of these scenarios—bar passage rates and ice cream profits—it will need to work over a wide range of inputs. The graph of the logistic function we showed earlier captures input data that ranges from `-10` to `10` crossing over from one category to the other at `0`. The GPA example needs a model that can capture input data that ranges from `0.0` to `4.0` crossing over at `3.0`; for the ice cream example, the model must capture input data from `0` degrees to `100` degrees crossing over at about `50` degrees.

This is where machine learning comes in. Training a logistic regression model involves shifting and stretching the logistic function to fit the shape of the data. We can do so by modifying the logistic function slightly. Rather than computing

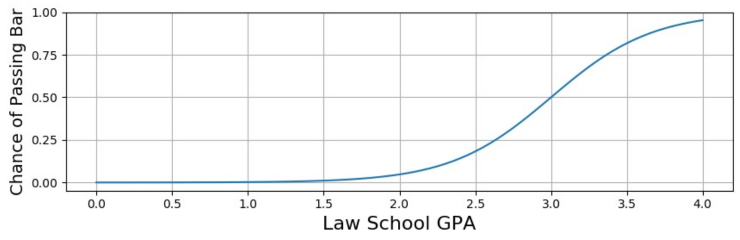
```
logistic_function(x)
```

we compute

```
logistic_function(w * x + b)
```

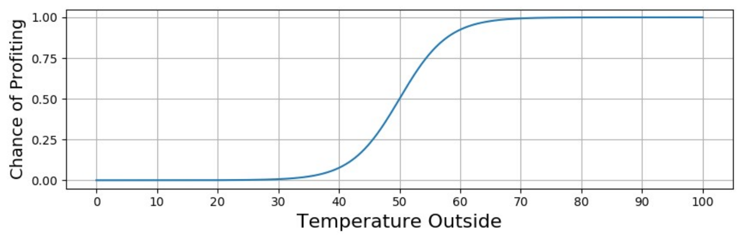
The variables `w` and `b` are the parameters of the logistic regression model. The `w` parameter controls how much the logistic function gets stretched or compressed in order to fit the range of the data. The `b` parameter controls how much the logistic function gets shifted to the left or right, changing where the crossover from Category 0 to Category 1 takes place. Learning from training data involves finding the right values of `w` and `b`.

For example, consider a logistic regression model for predicting whether a law student will pass the bar exam based on her law school GPA. By setting `w` to `3` (which compresses the logistic function) and setting `b` to `-9` (which shifts the crossover point to the right) we get the function `logistic_function(3 * x + 9)` whose graph is below:



This logistic function is a shifted and compacted version of the one we examined before. The function has been shifted to the right: it predicts that a student will have a 50% chance of passing the bar exam if she has earned a 3.0 GPA in law school. That is, the output reaches `0.5` when the input is `3.0`. In the original graph, this crossover point was instead at input `0`. The graph has also been compacted slightly. This model predicts that a student with a 1.0 GPA is almost certainly going to fail the bar exam, but that the chance of passing increases as the GPA does, reaching a 99% chance of passing when the student has a 4.0 GPA.

We could repeat a similar process for predicting ice cream sales. If we set `w` to `0.25`, which stretches the logistic function, and set `b` to `-12.5`, which shifts the function to the right, we get the function `logistic_function(.25 * x - 12.5)`, which has the following graph:



For temperatures of 30 degrees or less, this model predicts that there is a 0% chance that Larry will turn a profit (the logistic function outputs `0`). As the weather warms up into the 30s and 40s, the chance of profit increases. At 50 degrees, the chance of profiting reaches 50% (the logistic function outputs `0.5` on input `50`). As the temperature increases into the 50s and 60s, it becomes increasingly likely that Larry will profit. By the time it is 70 degrees outside, Larry is almost certain to profit (the logistic function outputs `1`). This model was created by changing `w` and `b` to shift the logistic function rightward (moving the crossover point to occur at 50 degrees rather than 0 degrees) and stretching the logistic function out so that it adequately covers a wide range of temperatures.

Summary. Let's consider logistic regression using our original framing of machine learning. We start with a model family: the logistic function. We then use training data to determine the exact parameters for the model. In the case of logistic regression, this involves setting `w` and `b` to the right values. We didn't discuss the exact algorithm for using the training data to find the best parameters because that is outside the scope of this book. The `scikit-learn` library, which we will use in the following section, does this automatically. Finally, once the model has been trained, we can predict how new inputs, like a GPA or a temperature, should be classified.

Multiple features. Notice that all of the examples we have considered so far have made predictions based on a single data point. For example, we predicted whether a law student would pass the bar based on her GPA alone. However, it is usually better to consider multiple features. We might also wish to take into account whether she took a bar prep class or the amount of time she has spent studying. Each of these features can be added to our logistic regression model:

```
logistic_function(w1 * gpa + w2 * prep + w3 * studying + b)
```

The variables `gpa`, `prep`, and `studying` are the features corresponding to the student's GPA, whether she took a prep class (`1` if she did and `0` if she didn't), and the number of hours she spent studying for the exam. Each of these features gets its own separate `w`. The model now has four parameters that have to be learned, `w1`, `w2`, `w3`, and `b`. This is a perfectly reasonable logistic regression model, and—since it has more data to work from—it is likely to be more accurate. However, this model is difficult to draw on a graph: since it has three separate inputs, it would require a four-dimensional graph to represent, well beyond the capabilities of the two-dimensional paper on which this book was written.

Most real-world logistic regression models have more than one (and often dozens) of features. For example, a logistic regression model operating on a picture of a cat might have hundreds of features, one for each pixel of the image. Although this point is subtle, it will be vitally important when we discuss neural networks.

04

Unknown

18.2.3 Logistic Regression in Python (Scikit Learn)

In practice, logistic regression involves two major steps:

1. Assemble training and test data.

2. Train a logistic regression model.

As it turns out, the `scikit-learn` library makes the second step so easy that we will spent much of our time in this chapter focusing on the first step.

The Supreme Court Database. The Supreme Court Database is a collection of carefully labelled data about Supreme Court cases dating back to 1791. It includes detailed information about cases that the Supreme Court heard, how they arrived at the Supreme Court, the issues they covered, and how each justice voted. Our goal in this section will be to explore whether we can use logistic regression to develop a model of a justice's voting behavior that makes it possible to predict how she will vote in future cases.

The dataset we will use in this section can be obtained by visiting the Supreme Court Database at `http://scdb.wustl.edu` and downloading the MODERN Database, justice centered data, organized by Supreme Court citation, in CSV form. Download the file, unzip it to extract the CSV version of the database, and place it in the same folder as your Python programs. We are now ready to begin.

Before doing anything else, we need to import three libraries.

```
1  import sklearn.linear_model # Access sckit-learn's logistic
       library.
2  import csv # Process csv files.
3  import random
```

On line 1, we import scikit-learn's `linear_model` library, which contains the logstic regression library we will use later in this program. On line 2, we import the CSV library, which is necessary to load the Supreme Court Database into Python an extract its contents. On line 3, we import the `random` library, which will be useful for randomly shuffling our training data.

Now, we can import the database itself.

```
5  # Load the Supreme Court Database using a CSV DictReader.
6  scdb = csv.DictReader(open('scdb.csv'))
```

The first row of the CSV file contains the name of each column of the CSV file. This means that we can use a `csv.DictReader`, which will associate each value that it loads with the name of the column from which it came. Each row that this `csv.DictReader` loads will be in the form of a dictionary mapping a column name to the corresponding value in that row. Since this dataset has dozens of columns, it is valuable to be able to refer to them by name.

We can now begin assembling our training data. To do so, we create two lists.

```
8  # Variables to store the training data.
9  xs = []
10 ys = []
```

The first variable, `xs`, will store the inputs to the logistic function. The variable `xs` is a list of lists: each item that it stores is a list containing the features for one item. For example, one item in `xs` could be a list containing the lower court from which the case originated and the topic of the case. The second variable, `ys`, will store, for each case, the justice's vote—the answer that we want our logistic regression to output when it sees the corresponding input. When the justice votes for the petitioner, the value in `ys` will be `1`; when the justice votes for the respondent, it will be `0`.

We are now ready to begin gathering training data. To do so, we will iterate over all of the rows in the `csv.DictReader`. Each row describes a single justice's vote. For now, we will try making a model of Antonin Scalia's voting pattern, but we could later substitute any justice.

```
12 # Assemble the training data.
13 for row in scdb:
14     # If this isn't a Scalia vote, skip this row.
15     if row['justiceName'] != 'AScalia': continue
```

On line 13, we iterate over each row in the `csv.DictReader` using a for-loop. Once inside the loop, the first action we take is to skip any rows that describe votes taken by any justice other than Scalia (line 14). The Supreme Court Database has a scheme for naming each piece of information about a row of the database. You can see the complete scheme at `http://scdb.wustl.edu/documentation.php`. The name of the justice who voted is under the key `'justiceName'`. On line 14, if we find that the justice is anyone other than Scalia, we immediately skip to the next iteration of the loop (and hence the next vote in the database).

Our next step is to determine the outcome: did Justice Scalia vote for the petitioner or the respondent? This step is a little tricky. The database does not tell us this information directly. Instead, it tells us whether Justice Scalia voted with the majority and whether the petitioner won. We will need to put these two pieces of information together to determine which way Scalia voted.

```
17     # Determine whether Scalia voted with the majority.
18     if row['majority'] == '': continue
19     majority = row['majority'] == '2' # True if he did.
20 
21     # Determine whether the petitioner won.
22     if row['partyWinning'] == '2': continue
23     petitioner = row['partyWinning'] == '1' # True if so.
24 
25     # Determine whether Scalia voted for the petitioner.
26     if majority and petitioner: y = 1
27     elif majority and not petitioner: y = 0
28     elif petitioner and not majority: y = 0
29     else: y = 1
```

Lines 17 to 29 determine whether Scalia voted for the petitioner (in which case the outcome we want from our classifier, `y`, should be `1`) or the respondent (in which ase `y` should be `0`). First, we use the `'majority'` column. This column is sometimes empty, in which case line 18 skips that row. If it isn't empty, then a value of `'2'` means the justice voted with the majority and `'1'` means he voted with the minority. The boolean variable `majority` is `True` if he voted with the majority and `False` otherwise.

We follow a similar procedure on lines 22-23 to determine whether the petitioner won. If the `'partyWinning'` column is `'2'`, then the outcome was uncertain; in this case, line 22 skips this row. Otherwise, the column is `'1'` if the petitioner won and `'0'` otherwise. We save whether the petitioner won to the boolean variable `petitioner`.

Finally, lines 26-29 determine whether Justice Scalia voted for the petitioner or the respondent and save the result to the variable `y`. These four lines include the boolean logic necessary to make sense of the data we have collected thus far.

We are finally ready to begin extracting features of the data. We want to find features that we think will be helpful to the logistic regression. Since we want to make predictions about the future, these features should be information that would be known before the outcome of the case has been revealed. One feature that might prove useful is whether the lower court made a "liberal" or "conservative" decision. If the lower court made a "liberal" decision, then a conservative-leaning justice might be more likely to side with the petitioner and overturn the lower court's decision. Likewise, if the lower court made a "conservative" decision, then a conservative justice might be more likely to uphold the lower court's decision and side with the respondent. We acknowledge that determining whether a decision was "liberal" or "conservative" is subjective, but any data, however imperfect, might be useful to help our model learn.

```
31      # Feature: did the lower court make a liberal decision?
32      if row['lcDispositionDirection'] == '' or row['
            lcDispositionDirection'] == '3':
33          continue
34      x1 = int(row['lcDispositionDirection'])
```

On line 32, we check whether the column is missing or is set to *unspecified* (`'3'`). If not, then the column is set to `'1'` if the decision was conservative and `'2'` if the decision was liberal. We convert this string into an integer and save it as our first feature in the variable `x1`.

At this point, we will cease to collect more features for the moment. Before doing so, we should check whether this feature alone was useful. We therefore complete the body of the loop by saving the feature (`x1`) and the outcome (`y`) to the lists `xs` and `ys` respectively.

```
36      # Save to the training data.
37      xs.append([x1])
38      ys.append(y)
```

We are now done with the body of the loop. The next lines of code can assume that the loop has fully assembled our training data into `xs` and `ys`, meaning we are ready to proceed with machine learning. As a final step, we will print the number of votes represented in our training data.

```
40  print('Votes in training data: {0}'.format(len(ys)))
```

When we run this program, the following output will print:

```
Votes in training data: 2739
```

Training a logistic regression model. Our training data is now stored in two lists, `xs` and `ys`. Each item in `xs` is a list of features corresponding to a particular vote; right now, these feature lists contain one item. Each item in `ys` is how Justice Scalia voted: `1` if for the petitioner and `0` if for the respondent.

Our first step when performing machine learning is to separate these lists into training data and test data. Recall from the first section of this chapter that we need to set aside some data that our model hasn't yet seen to evaluate how it performs.

```
42  # Shuffle the training data.
43  shuffles = zip(xs, ys)
44  random.shuffle(shuffles)
45  xs = []
46  ys = []
47  for shuffle in shuffles:
48      xs.append(shuffle[0])
49      ys.append(shuffle[1])
50
51  # Set aside 20% of the data as test data.
52  test = int(.8 * len(ys))
53  xs_train = xs[:test]
54  xs_test = xs[test:]
55  ys_train = ys[:test]
56  ys_test = ys[test:]
```

We want to set aside 20% of our dataset to be test data. Before we can do that, we need to put our dataset in random order. If we were to simply pick the test data to be the last 20% of the items in our dataset, then our test data would contain the 20% of votes that took place most recently. It is possible that the more recent votes look different than the older votes, meaning our test set has different behavior from our training set.

Lines 43 to 49 put the training data in random order. This is a surprisingly delicate process, since we want to make sure that each entry of `xs` is still paired with the corresponding entry of `ys`, even after randomizing the order of the two lists. To do so, we zip the lists into one list of tuples (line 43) and then shuffle that list of tuples (line 44). Lines 45-49 then separate these tuples back into two distinct lists.

Lines 52 to 56 slice `xs` and `ys` into separate lists for testing and training, with 80% of the data for training and 20% of the data for testing. Line 51 determines the index at which that 80% break occurs, and lines 53 to 56 perform the slicing.

After all of that preparation, we are ready to train the model. Doing so takes only a couple of lines of code.

```
58 # Create the model.
59 model = sklearn.linear_model.LogisticRegression()
60
61 # Fit the model to the data.
62 model.fit(xs_train, ys_train)
63
64 # Determine how accurate the model was on the test data.
65 accuracy = model.score(xs_test, ys_test)
66 print('Accuracy: {0}'.format(accuracy))
```

Line 59 creates a `LogisticRegression` object. Line 62 trains that object by calling the `fit` method with the training inputs (the first argument) and the expected training outputs (the second argument). Finally, line 65 uses the `score` method to assess how accurately the model performed on the test inputs and outputs.

Notice that training and predicting with a logistic regression model took only three lines of code. The vast majority of our program involved reading data, extracting features, and putting the data in the correct form to be processed by the machine learning model. This experience is reflective of data analysis and machine learning in the real world: far more time goes into data preparation than to actual modeling and analysis.

Although we did not use this method in the code above, the `predict` method takes as its sole argument a list of features and outputs the model's prediction (`0` or `1`). For example, to see how the model predicts Scalia would vote had the lower court made a liberal decision, we could write:

```
model.predict([2])
```

(Recall that `2` is the Supreme Court Database's code for "liberal" for the data about how the lower court decided.

Returning to the task at hand, we can run the program and see the following output.

```
Votes in training data: 2739
Accuracy: 0.651459854015
```

In other words, the model correctly predicted Justice Scalia's vote about 65% of the time on the test data. Since we randomly select the training and test data, this number will likely vary between 60% and 70% depending on luck.

At first glance, this number is pretty good. Using only a single feature, we were able to predict Justice Scalia's vote two thirds of the time.

However, it turns out that there was an even simpler strategy that would have yielded equivalent results. In practice, justices tend to vote in favor of overturning lower courts more often than upholding lower courts. In other words, a simpler alternative model is that we always guess that Justice Scalia will vote for the petitioner. How accurately does this model perform?

```
68 # Count the number of votes for the petitioner in the test data.
69 for_petitioner = 0.0
70 for vote in ys_test:
71     if vote == 1:
72         for_petitioner += 1
73
74 simpler_accuracy = for_petitioner / len(ys_test)
75 print('Accuracy of simpler model: {0}'.format(simpler_accuracy))
```

Lines 69 to 72 count the number of votes for the petitioner in the test set. Lines 74 and 75 determine the overall fraction of the test set that involves votes for the petitioner and prints that number.

```
Votes in training data: 2739
Accuracy: 0.667883211679
Accuracy of simpler model: 0.583941605839
```

This data shows that our logistic regression model does achieve about an 8% gain over the simpler model, so we can confirm that we have made some progress.

Adding more features and iterating. It is always desirable to do even better. The Supreme Court Database offers one other feature that seems interesting: whether the majority ruled in a "liberal" or "conservative" way. We can use this information to determine whether the petitioner represented the "liberal" or "conservative" side. Since many justices tend to consistently vote for the "liberal" or "conservative" side, our model may be able to learn Justice Scalia's tendencies and better predict his vote.

We can build another feature. In the original Python file we have developed, the code below would be inserted at line 35.

```
    # Feature: is the petitioner the liberal or conservative side?
    if row['decisionDirection'] == '' or row['decisionDirection']
        == '3':
        continue
    liberal = row['decisionDirection'] == '2'

    if petitioner and liberal: x2 = 1
    elif petitioner and not liberal: x2 = 0
    elif liberal and not petitioner: x2 = 0
    else: x2 = 1
```

These nine lines create a new feature, `x2`, which has the value `1` if the petitioner was the liberal side and `0` if the petitioner was the conservative side. The first three lines of this code block throw out any rows where this column was either empty or unspecified (`'3'`). The fourth line creates a boolean variable that is `True` if the majority decision was liberal and `False` otherwise.

The last four lines of this code block perform the logic to determine whether the petitioner was the liberal or conservative side. Recall that the variable `petitioner` is `True` if the petitioner won or `False` otherwise.

Our last step is to update the former line 37 to include both features.

```
37      xs.append([x1, x2])
```

We can now run the model to see how well it performs.

```
Votes in training data: 2737
Accuracy: 0.647810218978
Accuracy of simpler model: 0.582116788321
```

Unfortunately, this additional feature didn't seem to provide any improvement over the sole feature we had before. Within the framework we have developed in this section, one could easily experiment with many other features, including the circuit or state at which the case was previously decided and the subject matter of the case. It is possible that some combination of these features will yield better results, or that a more powerful kind of model (like a neural network) is necessary to model justices more accurately.[1]

This process of iteratively trying new features or different classes of models is a key part of the machine learning workflow. It is very rare for the first attempt at choosing features or selecting a model class to succeed. Instead, building a model is an gradual process of exploration of which we performed two steps in this section.

18.3 Neural Networks

Considering the hype ascribed to neural networks, it may surprise you to learn that they are merely a small embellishment of the logistic regression models we discussed in the previous section. You now have all of the conceptual knowledge you need to understand neural networks. In this section, we will introduce the notion of a neural network and show you how to create and train a model

[1]There is a rich body of research on predicting Supreme Court votes, much of which does indeed use more sophisticated models and data than we had available here. This FiveThirtyEight article summarizes the models and datasets people have applied to predict Supreme Court votes:
`https://fivethirtyeight.com/features/why-the-best-supreme-court-predictor-in-the-world-is-some-random-guy-in-queens/`

that classifies handwritten digits. This learning task is a popular choice for introductory tutorials into neural network development.

This chapter's standard caveat—that machine learning is a vast subject we can't possibly hope to cover in a single chapter—particularly applies to neural networks. Neural networks are especially unwieldy, and even experts often struggle to get them to learn effectively. Many experts will admit that training a neural network is more art than science. Furthermore, the state of the art is changing rapidly. Neural networks are still a topic of active research, so best practices are constantly evolving and this week's conventional wisdom is quite literally often out of date by next week.

With all of that said, neural networks are the foundation for all of today's most successful artificially intelligent systems, so it is vital to understand how they work and how to create them. This knowledge provides valuable insight into an increasingly pervasive technology that is often misrepresented or misunderstood in popular portrayals. Moreover, for those readers who are interested in delving into the vast, developing, and chaotic world of neural networks, this chapter provides a useful jumping off point.

For all of their flaws, neural networks are a demonstrably effective technique for taking on problems that, in prior decades, seemed far beyond the grasp of computer science. Not only are they here to stay, but they are likely to further grow in importance as we begin to fill in the myriad gaps in our scientific understanding of how they work. For that reason, we close out this chapter (and this book) by introducing the most exciting topic in modern machine learning: neural networks.

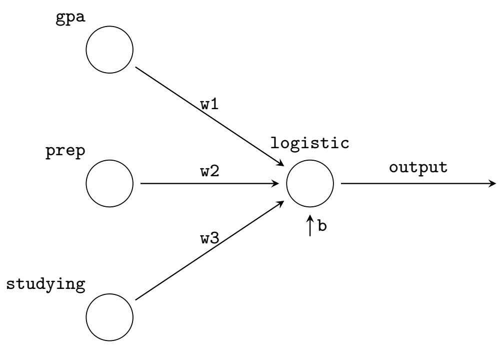
18.3.1 Neural Networks in Concept

From logistic regression to neural networks. The story of neural networks begins with logistic regression. Recall an example from our discussion of logistic regression: we want to predict whether a law student will pass the bar exam given three features: her law school GPA, whether she took a bar prep class, and how many hours she spent studying. Our logistic regression model for this learning task took the following form:

```
logistic_function(w1 * gpa + w2 * prep + w3 * studying + b)
```

This model takes three features: the student's law school GPA (`gpa`), whether the student took a bar prep class (`prep`), and the number of hours that the student spent studying (`studying`). The model also has three parameters, `w1`, `w2`, and `w3`, that control the extent to which each of these features is taken into account by the model. A fourth parameter, `b`, determines how much the logistic function will be shifted to the left or right. Training this model involves finding values for these four parameters that make accuracy as high as possible.

So far, all of this discussion is a review of Section 18.2. Below, we've rewritten the structure of the model as a flow chart.



The circles along the left from top to bottom are the three features from before: `gpa`, `prep`, and `studying`. The circle in the middle represents the logistic function. The arrow exiting from the right of the circle in the middle represents the output of the logistic function. This diagram is meant to represent the exact same logistic regression model as we just reintroduced. Let's see how it works.

The circle in the center representing the logistic function has four incoming arrows. One arrow goes from `gpa` to the logistic function and is labelled with `w1`. This arrow represents multiplying the `gpa` feature by its parameter `w1`, i.e., `gpa * w1`. Another arrow travels from `prep` to the logistic function and is labelled with `w2`, representing the value `prep * w2`. Similarly, the arrow from `studying` to the logistic function labelled with `w3` represents the value `studying * w3`. A fourth arrow labelled with `b` also points to the logistic function.

The four values that point to the logistic function are added together, producing the quantity

```
gpa * w1 + prep * w2 + studying * w3 + b
```

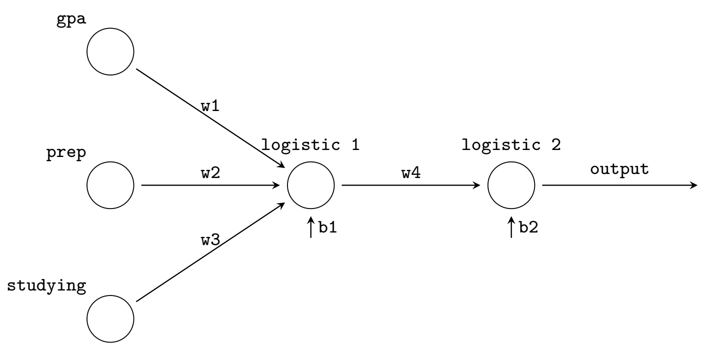
and then run through the logistic function, producing the output

```
logistic_function(w1 * gpa + w2 * prep + w3 * studying + b)
```

This output is represented by the arrow labelled `output` travelling rightward from the `logistic` circle.

This diagram is just the logistic regression equation rewritten in a different, visual form. It is identical to the formula we showed before. This diagram is useful because it shows the way data flows through the logistic regression. For example, this diagram shows how the `gpa` feature flows through the parameter `w1` before being combined with the other features and passed through the logistic function and how the output is derived from all of the features and parameters. Before moving on, study this diagram carefully to make sure you fully understand it. This diagram is your first neural network.

The key, counterintuitive insight behind neural networks is that it can be useful to perform logistic regression more than once. We can look at the output of logistic regression as just another feature. This new feature can, itself, be run through another logistic regression to produce another output. The diagram below shows this idea.



Just as before, our three features are the circles along the left. They are multipled by their corresponding `w`'s, added to the parameter `b1`, and fed into the logistic function (the circle labelled `logistic 1`). The result of this first logistic function, which we'll call `a1`, can be written as

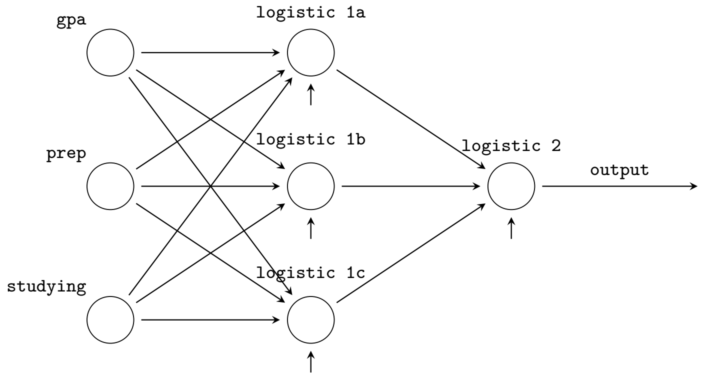
```
a1 = logistic_function(gpa * w1 + prep * w2 + studying * w3 + b1)
```

Before, `a1` was the output of our model. Here, however, the output of this first logistic function is the input to another logistic function. It is multiplied by another parameter, `w4`, added to the parameter `b2`, and run through another logistic function (the circle labelled `logistic 2`). The output of this second logistic function is the output of our model. In Python,

```
a2 = logistic_function(a1 * w4 + b2)
```

This model now has six parameters: `w1`, `w2`, `w3`, and `b1` just as in the previous model, and `w4` and `b2` from the additional logistic function.

This is the key insight behind neural networks: we can compute many logistic regressions and wire them together to create more sophisticated models. The previous diagram shows that we can arrange these logistic regressions in sequence, one after the other. We can also compute multiple logistic regressions in parallel and feed them into another logistic regression:



This diagram has four logistic regressions. Three of the logistic regressions, the circles labelled `logistic 1a`, `logistic 1b`, and `logistic 1c`, are all computed in parallel. Each of the three features (`gpa`, `prep`, and `studying`) has an arrow to each of the the three logistic regressions, making nine arrows in total between the three features and the first three logistic regressions. Each of these arrows has its own separate `w` (which we have hidden to make the diagram readable).

In Python, the output of `logistic 1a`, which we will call `a1a`, is computed as follows:

```
a1a = logistic_function(gpa * w1a + prep * w2a + studying * w3a +
     b1a)
```

This formula is a standard logistic regression with parameters `w1a`, `w2a`, `w3a`, and `b1a`.

The output of `logistic 1b`, which we call `a1b`, can be computed in similar fashion.

```
a1b = logistic_function(gpa * w1b + prep * w2b + studying * w3b +
     b1b)
```

This is yet another standard logistic regression with its own parameters separate from the other logistic regressions.

The output of `logistic 1c` is similar.

```
a1c = logistic_function(gpa * w1c + prep * w2c + studying * w3c +
     b1c)
```

The outputs of these logistic regressions, `a1a`, `a1b`, and `a1c`, are then fed into yet another logistic regression, `logistic 2`. This logistic function has its own set of parameters, `w1`, `w2`, `w3`, and `b`:

```
output = logistic_function(a1a * w1 + a1b * w2 + a1c * w3 + b)
```

The result of this final logistic regression is the output of the overall model. This model is yet more complicated: it has four logistic regressions and 16 parameters.

Defining a neural network. A neural network computes logistic regression (or other similar functions) on combinations of features in an iterative fashion. These functions are arranged into a network (hence the name). The process of training involves searching for parameters that improve the overall accuracy of the model.

What advantage does a neural network have over logistic regression? Why is it any better to compute logistic regression four times rather than just once? In the previous example, we computed three seemingly identicial logistic regressions of the input features—why is this any better than doing it once? Through the process of training a neural network, each of these logistic regressions will start to specialize. In other words, since these logistic regressions are each just a small part of a much larger model, they don't have to solve the entire problem on their own. Instead, they can learn intermediate information that doesn't solve the problem entirely but serves as a step along the way to the solution. This intermediate information—the outputs of the first three logistic regressions—are then fed into another logistic regression that can take advantage of this work to solve the remainder of the problem.

When the model is first created, all of the parameters are set to random values. This means that, even though the first three logistic regressions (`logistic 1a`, `logistic 1b`, and `logistic 1c`) are configured in the same way, they will initially produce completely different outputs. Initially, these outputs are likely to be nonsense. However, as the overall network learns, these logistic regressions will learn to output useful information as their parameters are slowly improved through the process of training. Since they were initially set to completely different, random values, they will learn to generate different kinds of intermediate information. The bottom line is that three logistic regressions will learn three different pieces of useful information, making the final logistic regression's job easier.

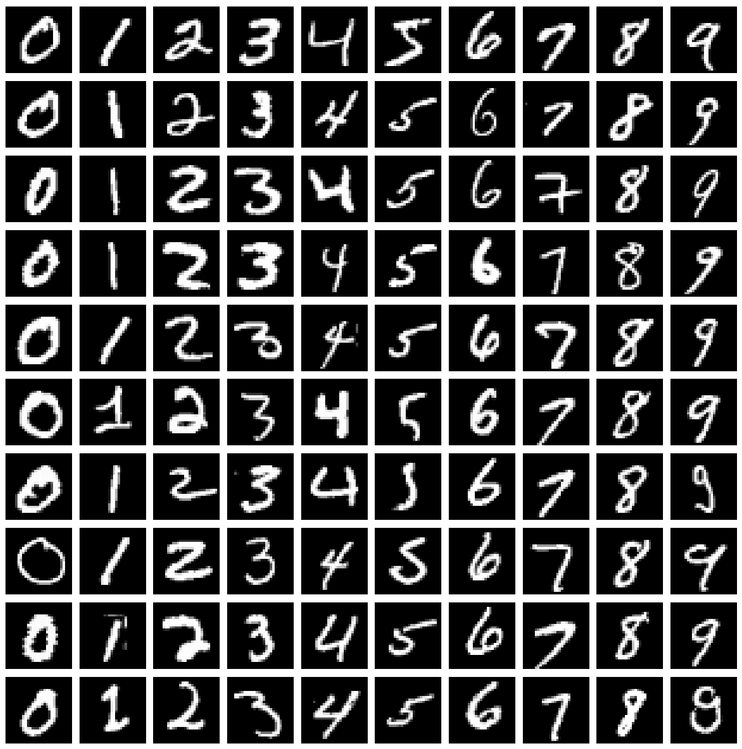
The networks we have considered so far are miniscule by modern standards. The network we will use to classify handwritten digits has 410 logistic regressions and nearly 270,000 parameters. Even this network is considerd tiny: industrial-strength networks for image classification or facial recognition might have tens of thousands of logistic regressions and billions of parameters. These gigantic models can take months to train. However, as you now know, the fundamental building block of all of these networks is logistic regression (or variations thereof).

Terminology. The field of neural networks has developed its own lexicon for describing each of the components we have introduced so far. The features of the original data (like `gpa`, `prep`, and `studying`) are known as inputs. Each of the parameters that the features are multiplied by (the `w`'s) are known as weights. The additional `b` values are known as biases. The logistic regressions, which we have drawn as circles, are known as units, nodes, or neurons (we will use the term unit here). Although all of the units we have considered so far compute the logistic function, many neural networks replace the logistic function with other functions that seem to work better in certain contexts. The particular function that a unit computes is called its activation function.

The units of a neural network are typically arranged into layers. For example, the network we showed previously has three layers. The features (the circles for `gpa`, `prep`, and `studying` on the far left) are known as the input layer. The column of units that comes next (`logistic 1a`, `logistic 1b`, and `logistic 1c`) are the first layer. Every feature in the input layer is connected to every unit in the first layer. When two layers are connected in this fashion, they are said to be fully-connected. (In more exotic neural network configurations, the layers are not fully connected, however we will only consider fully-connected layers in this chapter.) The third column, which contains just the unit `logistic 2`, is known as the second layer. Since it is the last layer, it is also referred to as the output layer because its output is the output of the entire model.

Most neural networks have several layers of units between the input layer and the output layer. The layers between the input layer and the output layer are known as hidden layers, as they connect to neither the output nor the input. Our handwritten digit classification network will have two hidden layers, one with 300 units and one with 100 units. Bigger networks for image classification might have between ten and twenty hidden layers, and researchers are experimenting with networks that have tens or hundreds of hidden layers. Networks with many layers are said to be deep; the term deep learning refers to performing machine learning with a deep neural network. There is no threshold for when a network goes from being "shallow" to being "deep," so the term "deep learning" has no practical significance beyond marketing.

Handwritten digit recognition. The National Institute of Standards and Technology (NIST) prepared a dataset containing thousands of pictures of handwritten digits. Some of the writers had excellent handwriting and the digits are easy to decipher. Others were less careful. Several early pioneers in neural networks research selected 70,000 pictures from this collection and compiled the MNIST dataset (`http://yann.lecun.com/exdb/mnist/`). Each of these pictures is a square (28 pixels by 28 pixels, making 784 pixels in total) containing a grayscale image of one of these digits. The machine learning problem is as follows: correctly identify as many of these digits as possible. Below are several examples images:



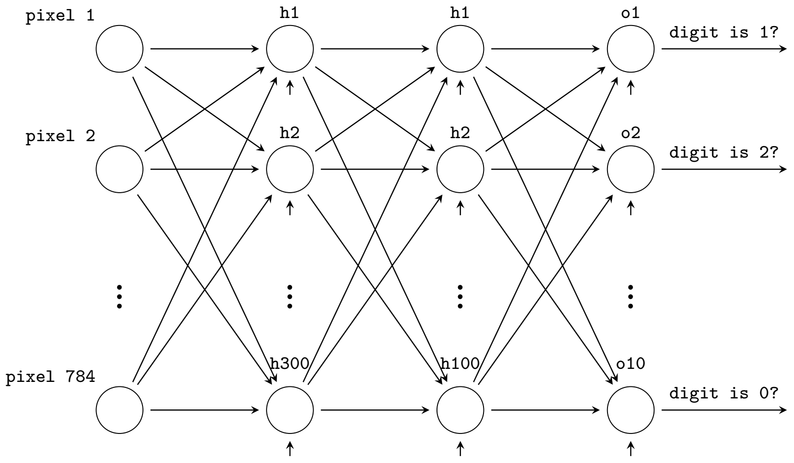
At first glance, this is a more challenging machine learning task than any we have encountered so far. However, we can apply the same methodology as we have in the past.

First, what is our task? We want to develop a model that can predict which digit (0 through 9) is contained within an image. This is a classification task. However, this classification problem is slightly harder than those we have considered in the past. All of the classification tasks we have discussed thus far have asked models to choose between two categories (will or won't pass the bar, will or won't be a profitable day for ice cream, will or won't vote for the petitioner). Here, the model must choose between ten categories: one for each digit that the image could contain.

Next, what are our features? Each picture in MNIST is made up of 784 pixels arranged in a 28 x 28 square. This pixel has a value between 0 and 1 representing its intensity from darkest (0) to lightest (1). Each of these pixels is one feature, meaning every image has 784 features.

Next, what is our training and test data? The MNIST dataset provides 70,000 images of digits. 60,000 of these images have been designated as training data and 10,000 have been designated as test data.

Now comes the hard part: what is our model family? We will use a neural network, but one that is much bigger than anything we have considered thus far. The input layer of this network has 784 units—one for each feature of an image. The network has two hidden layers. The first hidden layer has 300 units and the second hidden layer has 100 units. The output layer of the network has ten units—one for each possible classification decision we can make. All of the layers are fully connected, meaning that each unit is connected to all of the units in the layer before it. Using the same diagrams as before, our network architecture is as follows:



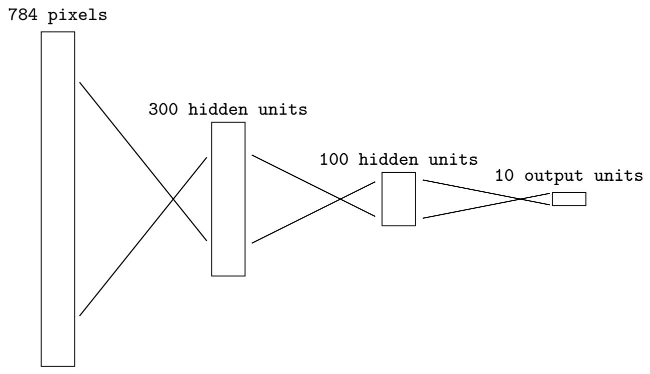
The left column represents the input layer, which has 784 units containing the values of the 784 pixels in each image. Only three units are drawn; the elipses between `pixel 2` and `pixel 784` signify the remaining 781 pixels that did not fit on this page. Each of these input units is connected to every unit in the second column from the left, which represents the first hidden layer. These units are labeled `h1` through `h300`, with 297 of the units represented by ellipses. These units multiply their inputs by the corresponding weights, add a bias, and compute the logistic function. Since there are 784 units in the input layer and 300 units in the second hidden layer, there are 235,200 weights connecting the two layers.

Each of the 300 units in the first hidden layer is connected to each of the 100 units in the second hidden layer (the third column from the left). Each unit in the second hidden layer multiplies the outputs of the first hidden layer by the corresponding weights, adds a bias, and computes the logistic function. Since there are 300 units in the first hidden layer and 100 units in the second hidden layer, there are 30,000 weights between the two.

Finally, each of the 100 units in the second hidden layer is connected to the ten units in the output layer (the last column on the right). There are 1000 weights between these two layers. The units in the output layer do not compute the logistic function. Instead, they compute a very similar function called the softmax function, which is like a ten-way logistic function. The softmax function ensures that the values of the ten output units add up to 1. Each output unit can be thought of as outputting the network's confidence that the image contains a particular digit. For example, if the first output unit's value is `0.5`, the second output unit's value is `0.35`, and the fifth output unit's value is `0.15`, then the network thinks there is a 50% chance the digit is a 1, and 35% chance the digit is a 2, and a 15% chance the digit is a 5. In an ideal world, the network would be 100% sure that the digit contained the right number; that is, if the network is fed a picture of an 8, we would hope that the eighth output unit would have the value `1.0` and all the other output units would have the value `0.0`. In reality, this is rarely the case, and we say that the network's decision is the digit to which it gives the highest probability.

Reread the previous paragraph carefully—the concepts are subtle but very important.

Before we continue, here is one other way of summarizing the network.



Each vertical bar represents the units in a layer. The X patterns represent the fact that the layers are fully-connected to one another. The leftmost bar represents the 784 input units representing the 784 images in the input layer. The second bar from the left represents the 300 units in the first hidden layer. They are fully connected to the input layer. The third bar from the left represents the 100 units in the second hidden layer, which are fully connected to the first hidden layer. Finally, the bar on the right represents the ten output units, which are fully connected to the second hidden layer. The bars are roughly in proportion to the sizes of the layers, giving you a sense for the way the network distills the massive amount of information contained in the raw pixels down to the ten outputs.

Training a neural network. Now that we have a network architecture, we need to determine out how to train it. A neural network is trained by repeating the following step over and over again, tens of thousands of times, until the network performs accurately:

1. Show the network an image from the training set.

2. Examine the output that the network produced.
3. Compare the output that the network produced to the correct output. For example, the network might say it was 60% certain that the image contained a 1 and 40% certain that the image contained a 7. However, if the image contained a 9, then the correct output would have been to be 100% sure that the image conatined a 9.
4. Slightly tweak the network's parameters so that its output on this image would have been closer to the correct output.

By repeating this operation over and over again, the parameters will slowly find their way toward values that accurately classify handwritten digits. Each repetition of these steps is known as a training iteration.

In practice, updating the parameters after a training iteration is very slow.[2] Rather than show the network one image, we typically show it a small mini-batch of images (typically 50 to 100 images), average the network's accuracy across those images, and then update the parameters based on that average accuracy. Doing so helps the network find good parameters in fewer training iterations and ensures that one idiosyncratic image doesn't throw off all of the network's parameters.

In general, the neural net training process can be understood as optimization: we want to find the set of parameters that minimizes the number and extent of mistakes that the network makes. There are many different ways of measuring the severity of a network's mistakes, although exploring that detail is beyond the scope of this book. The particular way of measuring the severity of a mistake is referred to as a loss function. Neural network training aims to find parameters that "minimize the loss," meaning the network makes fewer, smaller mistakes.

Summary. You now understand the basics of building and training neural networks. A neural network is simply many logistic functions (or other, related functions) chained together to create a large network with many parameters. These functions, which are known as units, are organized into layers that collectively analyze the input features and, layer by layer, put together the information necessary to classify it. These networks can be exceedingly large: even the small network we introduced for handwritten digit recognition contains 270,000 parameters that need to be trained. Neural networks are trained by repeatedly showing them mini-batches of training examples and updating the parameters to improve the accuracy that the network would have obtained on those examples.

The information we have covered in this section is just the basics of neural networks. Neural networks are an exceedingly hot topic for research and engineering, meaning that new configurations, paradigms, training methods, and optimizations are constantly being introduced. This section has described the

[2]For the mathematically inclined, it involves computing the partial derivative of the difference between network's output and the correct output (specifically, the network's loss, which we will introduce momentarily) with respect to each of the network's 270,000 parameters individually.

basic foundations that will can serve as a jumping off point for further learning or as a body of knowledge for analyzing neural networks that appear in law and policy contexts. In the next section, we will conclude this chapter by building in Python the neural network we designed for handwritten digit recognition.

05

Unknown

18.3.2 Neural Networks in Python (Keras)

In this section, we will build the handwriting-recognition neural network that we described in the previous section. To do so, we will use a library called `keras`, which makes creating and training small neural networks exceedingly easy. In fact, we will import the data and create, train, and evaluate the network in less than 30 lines of Python (not including comments and whitespace). As was our experience in Section 18.2, it will take as much work to import and clean the data as it will to build, train, and use the model.

Before we can begin to build neural networks, you will need to install three Python libraries:

1. `numpy`, a high-performance numerical processing library. If you followed Section 18.2, you have already installed `numpy`.

   ```
   $ pip3 install numpy
   ```

2. `tensorflow`, a low-level library for building and training neural networks developed by Google.

   ```
   $ pip3 install tensorflow
   ```

3. `keras`, a high-level library that makes `tensorflow` dramatically easier to use.

   ```
   $ pip3 install keras
   ```

Importing data. Just as we experienced in Section 18.2, it will take a certain amount of work to simply get the MNIST dataset into the right form for our neural network.

First, we need to import two libraries.

```
import numpy
import keras
```

The `numpy` library is necessary for some of the numerical processing that we will need to do. The `keras` library will build, train, and evaluate our neural network.

We can now access the MNIST data. Since this dataset is so common for training and evaluating neural networks, it comes built into `keras`.

```
# Import the MNIST training data.
mnist = keras.datasets.mnist.load_data()
```

Line 5 asks `keras` to download the dataset if it has not been downloaded previously and to load it into the variable `mnist`. The first time you run this statement, it may take a couple of minutes for MNIST to download. The variable `mnist` is a tuple with two elements. The first element is the training set and the second element is the test set. We separate those two elements on lines 6 and 7.

```
 6  train = mnist[0]
 7  test = mnist[1]
```

The training set contains the 60,000 images in the official MNIST training set, and the test set contains the 10,000 images in the official MNIST test set. Both of these are, themselves, tuples with two elements. The first element is the input features (that is, the images themselves) and the second element is the digit written in that picture (that is, the output we want the network to produce on that image). On lines 8 to 11, we separate out this data.

```
 8  x_train = train[0]
 9  y_train = train[1]
10  x_test = test[0]
11  y_test = test[1]
```

We finally have all of the data we need to train the network. Unfortunately, neither the x-values nor the y-values are in the proper form we need for our network. We will begin with the x-values, which have two problems. First of all, the x-values have the intensity of each pixel (how light or dark it is) stored as an integer between `0` and `255`. It will be much easier on the network if this value is between `0.0` and `1.0`. In short, we need to divide every pixel of every image by `255`.

Thankfully, the `numpy` library will make this job easier. The x-values and y-values are not stored as normal Python lists. Instead, they are stored as a special `numpy` type called an array. These `numpy` arrays are very similar to Python lists, but they have several features that make them easier to work with in numerical settings. In this case, we can divide the entire array by `255` in one step.

```
13  # Put the x-values in the proper form.
14  x_train = x_train / 255
15  x_test = x_test / 255
```

If we try to divide a list by `255`, Python will crash with an error message. If we divide a `numpy` array by `255`, `numpy` will understand our intent and divide every element in that array by `255`.

The second problem with our x-values is that they have the wrong shape. Right now, `x_train` is a `numpy` array with 60,000 elements—one for each image in the training set. Each of these elements is a 28 x 28 `numpy` array containing the pixels of the image arranged in a square. All in all, the shape of this array is said to be 60,000 x 28 x 28, since it has 60,000 images that are each 28 x 28. However, our network needs these pixels arranged in a single list of 784 elements. In other words, we want the shape of this array to be 60,000 x 784. Thankfully, `numpy` arrays make this transformation easy as well.

```
16 x_train = x_train.reshape([60000, 784])
17 x_test = x_test.reshape([60000, 784])
```

Each `numpy` array has a method called `reshape` that will return a new array containing a copy of the original array with its elements rearranged into the shape you want. It takes as its argument a list of the dimensions that you want the array to be converted to. The `reshape` method keeps all of the values of the array in the same order but changes the dimensions, so you have to use it with care to ensure that it does what you intend. In this case, however, it does exactly what we want.

The x-values are now in the proper form, but we still have to fix the y-values. The problem here is that each element of `y_train` and `y_test` is the integer that the corresponding picture contains. For example, if the picture had an image of the number eight, then the corresponding y-value would be `8`. However, as we discussed in the previous section, we want the y-value to contain the percentages that each of the network's output units should produce. In other words, the y-value we want in this case should be a list of ten elements with a `1` at index 8 and a `0` everwhere else: `[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]`. Here, `keras` will bail us out. It has a function designed for exactly these circumstances.

```
19 # Put the y-values into the proper form.
20 y_train = keras.utils.np_utils.to_categorical(y_train)
21 y_test = keras.utils.np_utils.to_categorical(y_test)
```

The `keras.utils.np_utils.to_categorical` converts every element of `y_test` and `y_train` into the correct form. We are finally ready to train the model.

Building a neural network model. The `keras` library makes building a neural network very easy. First, we create a `keras.models.Sequential` object representing an initially empty network. We can then build the network layer by layer.

```
23 # Build the model.
24 model = keras.models.Sequential()
```

We can then add the first fully-connected hidden layer, which has 300 hidden units. The `keras` library refers to a fully-connected layer as a "dense" layer (since it is densely-connected to the previous layer.).

```
25 model.add(keras.layers.Dense(300, activation='sigmoid',
       input_shape=[784]))
```

Line 25 creates an object that describes the layer we wish to add to the network (a `keras.layers.Dense` object) and then adds it to the network by passing it as an argument to the `add` method of `model`. The first argument of the `keras.layers.Dense` function is the number of hidden units in the layer. As we discussed in the previous section, the first hidden layer has 300 units.

We also provide two keyword arguments: `activation` and `input_shape`. The `activation` argument specifies which activation function we want to use for this layer. Recall that an activation function is the function that the unit runs all of its inputs through after adding them together. In the model we discussed in the previous section, we always used the logistic function as our activation function. Here, we do the same thing. The sigmoid function is just another name for the logistic function; the term sigmoid is more popular in the neural network community.

The other keyword argument, `input_shape`, specifies the shape of each input that will be provided to this layer from the previous layer. Since the first hidden layer immediately follows the input layer, it should expect to see inputs with 784 elements. This argument has to be provided as a one-element list, since some kinds of networks work with two-dimensional inputs (like images). The `input_shape` argument only needs to be passed to the first hidden layer, since it has no other way of knowing how big the input is. Subsequent layers can figure this out by looking at previous layers.

We repeat the same exercise for the next hidden layer.

```
model.add(keras.layers.Dense(100, activation='sigmoid'))
```

The second hidden layer has 100 hidden units, and we change the argument accordingly. It still has a sigmoid activation function. This time, we do not need to provide an `input_shape` argument.

Finally, we create the output layer.

```
model.add(keras.layers.Dense(10, activation='softmax'))
```

This layer has ten units—one for each of the possible digits. As you'll recall from the previous section, this layer needs a special softmax activation function that ensures that the values of the output units always adds up to 1.

In four lines of Python, our network is now complete. The last step is to compile the network. The `keras` library is just a convenient way of writing networks that will be created and run in the lower-level `tensorflow` library. In other words, `keras` needs to compile the network we just created into a form acceptable to `tensorflow`.

```
model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
```

To compile the model, we use the `model` object's `compile` method. We need to pass this method several arguments that are necessary for the model to train correctly but that will nearly always be the same whenever you use `keras`. The `loss` argument specifies the loss function that the training process will use to measure whether the network has improved. In classification tasks, it is common to use a loss function known as cross entropy loss to which the argument `'categorical_crossentropy'` refers. We do not go into the details of cross entropy loss in this book, but it is a good choice in most classification settings.

The `optimizer` argument specifies how the training process should modify the network's parameters during each training iteration. The adam algorithm is a common choice, hence the argument `'adam'`.

Finally, the `metrics` argument specifies any additional information we want `keras` to tell us about the network as it trains. In our case, the most important metric is accuracy—how often did the network make the right choice.

Training the neural network model. We are now ready to train the neural network. Just as with `scikit-learn`, training takes only one line of code.

```
model.fit(x_train, y_train, batch_size=100, epochs=10, verbose=1)
```

The `fit` method tells `keras` to train the model. The first two arguments to the `fit` method are the training set's inputs and outputs. We also need to tell `keras` a few other pieces of information. The `batch_size` argument tells `keras` how many training images to consider in each training iteration; 100 is a reasonable choice.

The `epochs` argument tells `keras` how long to train the network. As `keras` trains the network, it will iterate over the entire training set until it runs out of images. Each pass through the training set is known as a epoch. The `epochs` argument tells `keras` how many times to iterate over the training set before training is complete; 10 epochs is a reasonable choice for this network and dataset.

Notice that, for both of the previous two arguments, we have simply told you what reasonable values were without giving you an explanation for why. The same is true for many of the numbers in this section: why does the network have two hidden layers of 300 and 100 units each? Why not more or less? Designing and training neural networks remains more art than science. Over the years, neural network engineers have found these numbers to be effective for training a model for MNIST. For other learning tasks on other datasets, different numbers have been found to work well. When you take on a new machine learning problem, you will have to find the right numbers for yourself. In many ways, neural networks are still in their infancy, and we have yet to develop a general set of principles for guiding neural network development beyond simple trial and error.

Returning to the task at hand, the last argument of the `fit` method, `verbose`, tells `keras` that we want it to give us detailed diagnostic information as the network trains.

When you call this method, the network will begin to train. Training a neural network is a complicated endeavor, and it will take anywhere from two to five minutes for your network to train. Since we used the `verbose` argument, `keras` will keep us posted on how training is progressing.

```
60000/60000 [==============================] - 5s 88us/step -
    loss: 0.5551 - acc: 0.8583
Epoch 2/10
```

```
60000/60000 [==============================] - 5s 76us/step -
    loss: 0.2184 - acc: 0.9361
Epoch 3/10
60000/60000 [==============================] - 4s 73us/step -
    loss: 0.1593 - acc: 0.9529
Epoch 4/10
43600/60000 [====================>.........] - ETA: 1s - loss:
    0.1263 - acc: 0.9629
```

The output during training should look something like the above. Each line is a progress bar showing how many training examples `keras` has used during the curent epoch. At the end of each epoch, it outputs two values, the `loss` and `acc` (i.e., accuracy) as computed on the training set. With each epoch, the loss should go down as the network makes fewer mistakes. Correspondingly, the accuracy should go up as the network makes fewer mistakes. By the end of the first epoch, the network has already reached 85% accuracy on the training set. By the end of the fourth epoch, it has reached 96%.

While accuracy on the training set is important, it does not tell us anything about how the network will perform on new inputs. After all, if the network simply memorized the training set, it could get perfect accuracy on the training set while being unable to make any predictions about new inputs. This is where the test set comes in. The last step in the machine learning process is to evaluate our model on inputs it has never seen before—the test set. The `model` object has an `evaluate` method for doing just that.

```
32 # Evaluate the model on the test set.
33 results = model.evaluate(x_test, y_test)
34 print('Test loss: {0}'.format(results[0]))
35 print('Test accuracy: {0}'.format(results[1]))
```

The `evaluate` method takes two arguments: the test inputs (`x_test`) and the expected outputs (`y_test`). It returns a tuple with two elements. The first element is the loss on the test set, and the second element is the accuracy on the test set. Lines 34 and 35 print both values. Below is the output that `keras` produces after training and testing.

```
60000/60000 [==============================] - 5s 89us/step -
    loss: 0.5815 - acc: 0.8506
Epoch 2/10
60000/60000 [==============================] - 5s 78us/step -
    loss: 0.2187 - acc: 0.9366
Epoch 3/10
60000/60000 [==============================] - 5s 86us/step -
    loss: 0.1609 - acc: 0.9525
Epoch 4/10
60000/60000 [==============================] - 5s 89us/step -
    loss: 0.1231 - acc: 0.9632
Epoch 5/10
```

```
60000/60000 [==============================] - 5s 91us/step -
    loss: 0.0958 - acc: 0.9726
Epoch 6/10
60000/60000 [==============================] - 5s 90us/step -
    loss: 0.0768 - acc: 0.9776
Epoch 7/10
60000/60000 [==============================] - 5s 90us/step -
    loss: 0.0612 - acc: 0.9818
Epoch 8/10
60000/60000 [==============================] - 5s 89us/step -
    loss: 0.0501 - acc: 0.9856
Epoch 9/10
60000/60000 [==============================] - 5s 85us/step -
    loss: 0.0402 - acc: 0.9885
Epoch 10/10
60000/60000 [==============================] - 5s 84us/step -
    loss: 0.0328 - acc: 0.9910
10000/10000 [==============================] - 1s 61us/step
Test loss: 0.06755954547743313
Test accuracy: 0.9786
```

At the end of training, the accuracy on the test set is 97.86%, which is excellent for a seemingly tricky learning task. However, notice that at the end of the last epoch, the accuracy on the training set is 99.10%. Why is the accuracy on the training set measurably higher than on the test set? By the end of the tenth epoch, the model has seen every example in the training set. It has learned a model that is exceedingly good at classifying those examples. However, it has gotten so used to the training set that it is unable to replicate that performance on new examples. This is a phenomenon called overfitting, where the model learns the nuances of the training set so closely that it becomes slightly worse at classifying new examples.

Before we close this section, there is one final method that you will find useful. If you have new data and want to see the predictions that your model makes, you can use the `predict` method. It takes as its sole argument an x-value (in this case, an image), and produces its predicted y-value (in this case a list of ten elements containing the model's certainty about whether the image is of each digit).

```
model.predict(x)
```

With that, you now know all of the basics of creating, training, and testing neural networks with the `keras` library. There is a far larger world of neural network designs and techniques that you can explore, and you now have a solid jumping-off point from which to begin.

18.3.3 Challenges in Neural Networks

Neural networks date back to the dawn of modern computing. The initial concept that evolved into modern neural networks, known as the perceptron, was invented in the 1950s. Neural network development has proceeded in bursts since then, with the most recent resurgence, which began in 2011, leading to their current prominence.

In spite of these decades of study, we still have a poor understanding of the kinds of neural networks that are commonly in use today. In this section, we will explore why this is the case and how this manifests in several different categories of practical challenges.

In general, all of the challenges with modern neural nets trace back to their sheer scale. The distinguishing quality of the current generation of neural networks is that they involve exceedingly large models (millions of parameters) and trained on exceedingly large datasets (hundreds of millions of training examples). We currently do not understand how to translate such complicated models into high-level algorithms that make sense to humans.

As an analogy, consider asking Larry the lawyer for his favorite flavor of ice cream. He tells you that his favorite flavor is vanilla. Analogously, you can supply a picture to a neural net and it will tell you what object it contains.

Now suppose you wish to understand *why* Larry's favorite flavor is vanilla. Rather than simply tell you, Larry instead hands you a scan of his brain showing the connections between the tens of billions of its constituent neurons. Then he says, "Here is a map of my brain, the biological algorithm that decided I that I like vanilla. You figure out why it made that decision." At the moment, understanding a neural network's decisionmaking process is as difficult as understanding Larry's preference for ice cream by looking at the internal wiring of his brain.

As neural network-based artificially intelligent systems continue to proliferate into contexts where they can have immensely consequential impacts on people's lives, this situation poses a variety of concerning practical challenges. With that said, it is important to remember that this is simply the current state of research, not a permanent condition of using neural networks. We have been using neural networks for large scale machine learning applications for less than a decade, so in many ways this generation of the technology remains in its infancy. It is common for powerful new technologies to be applied long before they are fully understood, and neural networks are no exception. Although many of the challenges we discuss in this section may seem disquieting or even dire, the state of the art is developing rapidly and it is entirely possible that we may solve some or all of these problems in the near future.

Explainability and interpretability. Neural networks are increasingly tasked with making consequential decisions, for example, who gets hired and who gets credit. In many of these settings, we wish to know why a neural network made a particular decision. When a self-driving car inexplicably fails to apply the brakes at a red light, it is important to be able to understand why it did so.

As we have already insinuated, the decisions made by neural networks currently defy explanation. We do not know how to translate complicated connections between thousands of units into human-comprehensible insights about the network's decisionmaking process.

It is possible that a neural network has learned a simple algorithm that we simply do not know how to extract. It is also possible that a neural network has learned an algorithm that is very effective at solving the problem at hand but would seem counterintuitive to humans or, worse, is too sophisticated for a human to easily grasp. Finally, it is possible that the network has simply memorized the training set or has learned some heuristic that works on the training and test sets but doesn't work in general. Right now, we have no way of telling the difference between these three scenarios.

To add one last complication to the question of explaining a neural network's decisionmaking process, we have yet to clearly articulate what it means—in technical terms—to produce a "human-comprehensible explanation." Different kinds of explanations make sense in different contexts to different humans, and different circumstances may demand vastly different levels of detail. For example, consider one possible form of explanation for a neural network's decision: *The network had two hidden layers of 300 and 100 hidden units with sigmoid activation functions. It was trained on 60,000 images of handwritten digits for 10 epochs. Once the parameters were so trained, we fed it an image and it decided it contained the number three.* This certain explains why a network made a decision, but not in a fashion that is useful.

In summary, the sheer complexity of a neural network makes it difficult to understand why it made a particular decision. Right now, we have yet to even define what a "good" explanation would look like.

Debug-ability. Suppose you discover a flaw in a neural network. For example, whenever a self-driving car encounters a particularly tricky driving situation, it makes the wrong decision. You want to somehow "correct" this behavior in the neural network.

Right now, we are unable to reconfigure or "rewire" the neural network to perform these sorts of fixes, owing to the fact that we do not know how to understand the high-level algorithm (if any) reflected in the neural network's underlying circuitry. In other words, even if we find a bug in a neural network, we have no way to fix the network to remove this bug.

The only way to attempt to fix a network is to train it more (or to retrain it from scratch). The training set should be updated to include examples that show the network the correct decisions to make in circumstances where it is currently making mistakes. However, there is no guarantee that this corrective training will necessarily help the network avoid these mistakes. We do not fully understand the impact that training has on the network; it is possible that the network will ignore these new training examples or that the lessons that the network learns from these examples will come at the cost of forgetting other knowledge.

In summary, even when we find bugs in neural networks, we have no reliable way to fix them.

Reliability and security. Neural networks can fail in unexpected, catastrophic ways. For example, researchers have shown that neural networks can easily be fooled. Supose you have a picture that both you and the network agree contains a cat. This picture can be slightly modified in a carefully-calibrated way to create a new picture that looks unmodified to you but looks like a dog to the network.

In general, since we do not understand how to interpret the neural network's circuitry, it is possible that the network has learned bad behaviors that go completely undetected until they are triggered by the wrong input. In other words, a perfect storm of unlucky circumstances may trigger some catastrophic behavior that was previously invisible.

Worse, an attacker can exploit this fact to design bad behaviors and secretly embed them into the network. For example, an attacker can "poison" the network's training data: he can train the network underlying a self-driving car to suddenly accelerate every time it sees a stop sign with a special sticker attached to it. When the car sees a normal stop sign, it will behave as expected and this hidden behavior will go undetected. However, when the attacker puts one of these special stickers on a stop sign, the car will suddenly accelerate.

Bias. Neural networks (and all machine learning systems) aim to emulate their training sets. However, most training sets come from real-world data, and this data inherits the implicit biases surrounding its creation. Networks that learn from this biased data will, in turn, be biased. Right now, researchers are working to determine what it means for a network to be biased, how we can measure the extent of a network's bias, and how we can train networks in ways that eliminate bias.

18.4 Summary

This chapter attempted to distill an enormous topic down to several dozen pages. In this chapter, you learned the definition of machine learning, the high-level process that it follows, and two concrete techniques for performing machine learning. Logistic regression is a relatively simple method that can learn to separate inputs into two different categories. By connecting many logistic regressions together, you can build a neural network, a dramatically more powerful (and more complicated and unwieldy) technique for building models. This chapter gave you both a conceptual introduction to all of these topics and concrete Python knowledge that can serve as a foundation for further exploration.

With the end of this chapter, your journey through this text comes to a close. Over the preceding pages, you have investigated what it means to program, written your first Python programs, learned the core of the Python language,

Machine Learning

Court, Supreme

01 Unknown Unknown Page 7

7/9/2019 12:42

02 Unknown Unknown Page 8

7/9/2019 12:43

03 Unknown Unknown Page 9

7/9/2019 12:55

04 Unknown Unknown Page 14

7/9/2019 12:45

05 Unknown Unknown Page 32

7/9/2019 12:48