MACHINE LEARNING WITH TensorFlow

Nishant Shukla

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Machine Learning with TensorFlow
Version 6

For more information on this and other Manning titles go to
www.manning.com

welcome

Dear readers, I deeply appreciate your feedback throughout the months. I've taken the time to address each comment as you'll see with this latest MEAP update. Moreover, I've added many more figures, and slowed down when explaining more difficult concepts. Keep in touch with me on the book forums (machine-learning-with-tensorflow).

Thank you for investing in the MEAP edition of *Machine Learning with TensorFlow*. You're one of the first to dive into this introductory book about cutting-edge machine learning techniques using the hottest technology (spoiler alert: I'm talking about TensorFlow). You're a brave one, dear reader. And for that, I reward you generously with the following.

You're about to learn machine learning from scratch, both the theory and how to easily implement it. As long as you roughly understand object-oriented programming and know how to use Python, this book will teach you everything you need to know to start solving your own big-data problems, whether it be for work or research.

TensorFlow was released less than a year ago by some company that specializes in search engine technology. Okay, I'm being a little facetious; well-known researchers at Google engineered this library. But with such prowess comes intimidating documentation and assumed knowledge. Fortunately for you, this book is down-to-earth and greets you with open arms.

Each chapter zooms into a prominent example of machine learning, such as classification, regression, anomaly detection, clustering, and many modern neural networks. Cover them all to master the basics, or cater it to your needs by skipping around.

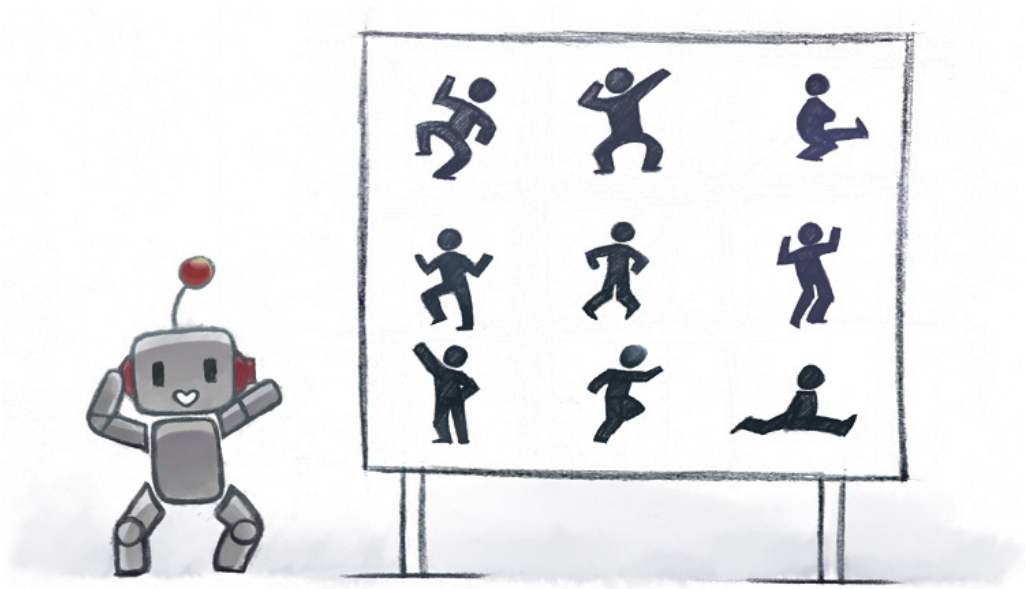
Keep me updated on typos or minor mistakes, since this book is undergoing heavy development. It's like living in a house that's still actively under construction; at least you won't have to pay rent. But on a serious note, your feedback along the way will be appreciated.

With gratitude,
—Nishant Shukla

brief contents

1

A machine-learning odyssey



This chapter covers

- Machine learning fundamentals
- Data representation, features, and vector norms
- Existing machine learning tools
- Why TensorFlow

Have you ever wondered if there are limits to what computer programs can solve? Nowadays, computers appear to do a lot more than simply unravel mathematical equations. In the last half-century, programming has become the ultimate tool to automate tasks and save time, but how much can we automate, and how do we go about doing so?

Can a computer observe a photograph and say "ah ha, I see a lovely couple walking over a bridge under an umbrella in the rain"? Can software make medical decisions as accurately as that of trained professionals? Can software predictions about the stock market perform better than human reasoning? The achievements of the past decade hint that the answer to all these questions is a resounding "yes," and the implementations appear to share a common strategy.

Recent theoretic advances coupled with newly available technologies have enabled anyone with access to a computer to attempt their own approach at solving these incredibly hard problems. Okay, not just anyone, but that's why you're reading this book, right?

A programmer no longer needs to know the intricate details of a problem to solve it. Consider converting speech to text: a traditional approach may involve understanding the biological structure of human vocal chords to decipher utterances using many hand-designed, domain-specific, un-generalizable pieces of code. Nowadays, it's possible to write code that simply looks at many examples, and figures out how to solve the problem given enough time and examples.

The algorithm learns from data, similar to how humans learn from experiences. Humans learn by reading books, observing situations, studying in school, exchanging conversations, browsing websites, among other means. How can a machine possibly develop a brain capable of learning? There's no definitive answer, but world-class researchers have developed intelligent programs from different angles. Among the implementations, scholars have noticed recurring patterns in solving these kinds of problems that has led to a standardized field that we today label as *machine learning* (ML).

Trusting machine learning output

Detecting patterns is a trait that's no longer unique to humans. The explosive growth of computer clock-speed and memory has led us to an unusual situation: computers now can be used to make predictions, catch anomalies, rank items, and automatically label images. This new set of tools provides intelligent answers to ill-defined problems, but at the subtle cost of trust. Would you trust a computer algorithm to dispense vital medical advice such as whether to perform heart surgery?

There is no place for mediocre machine learning solutions. Human trust is too fragile, and our algorithms must be robust against doubt. Follow along closely and carefully in this chapter.

1.1 Machine learning fundamentals

Have you ever tried to explain to someone how to swim? Describing the rhythmic joint movements and fluid patterns is overwhelming in its complexity. Similarly, some software problems are too complicated for us to easily wrap our minds around. For this, machine learning may be just the tool to use.

Hand-crafting carefully tuned algorithms to get the job done was once the only way of building software. From a simplistic point of view, traditional programming assumes a deterministic output for each of its input. Machine learning, on the other hand, can solve a class of problems where the input-output correspondences are not well understood.

Full speed ahead!

Machine learning is a relatively young technology, so imagine you're a geometer in Euclid's era, paving the way to a newly discovered field. Or, treat yourself as a physicist during the time of Newton, possibly pondering something equivalent to general relativity for the field of machine learning.

Machine Learning is about software that learns from previous experiences. Such a computer program improves performance as more and more examples are available. The hope is that if you throw enough data at this machinery, it will learn patterns and produce intelligent results for newly fed input.

Another name for machine learning is *inductive learning*, because the code is trying to infer structure from data alone. It's like going on vacation in a foreign country, and reading a local fashion magazine to mimic how to dress up. You can develop an idea of the culture from images of people wearing local articles of clothing. You are learning *inductively*.

You might have never used such an approach when programming before because inductive learning is not always necessary. Consider the task of determining whether the sum of two arbitrary numbers is even or odd. Sure, you can imagine training a machine learning algorithm with millions of training examples (outlined in Figure 1.1), but you certainly know that's overkill. A more direct approach can easily do the trick.

Input		Output
$x_1 = (2, 2)$	→	y_1 = Even
$x_2 = (3, 2)$	→	y_2 = Odd
$x_3 = (2, 3)$	→	y_3 = Odd
$x_4 = (3, 3)$	→	y_4 = Even
...		...

Figure 1.1 Each pair of integers, when summed together, results in an even or odd number. The input and output correspondences listed are called the ground-truth dataset.

For example, the sum of two odd numbers is always an even number. Convince yourself: take any two odd numbers, add them up, and check whether the sum is an even number. Here's how you can prove that fact directly:

> **For any integer *n*, the formula *2n+1* produces an odd number. Moreover, any odd number can be written as *2n+1* for some value *n*. So the number *3* can be written *2(1) + 1*. And the number *5* can be written *2(2) + 1*.**
>
> **So let's say we have two different odd numbers *2n+1* and *2m+1*, where *n* and *m* are integers. Adding two odd numbers together yields *(2n+1) + (2m+1) = 2n + 2m + 2 = 2(n+m+1)*. This is an even number because 2 times anything is even.**

Likewise, we see that the sum of two even numbers is also an even number: $2m + 2n = 2(m+n)$. And lastly, we also deduce that the sum of an even with an odd is an odd number: $2m + (2n+1) = 2(m+n) + 1$. Figure 1.2 visualizes this logic more clearly.

	even	odd
even	$2m + 2n = 2(m + n)$ even	$2m + (2n+1) = 2m + 2n + 1$ odd
odd	$(2m+1) + 2n = 2m + 2n + 1$ odd	$(2m+1) + (2n+1) = 2(m + n + 1)$ even

Figure 1.2 This table reveals the inner logic behind how the output response corresponds to the input pairs.

That's it! With absolutely no use of machine learning, you can solve this task on any pair of integers someone throws at you. Simply applying mathematical rules directly can solve this problem. However, in ML algorithms, we can treat the inner logic as a *black box*, meaning the logic happening inside might not be obvious to interpret.

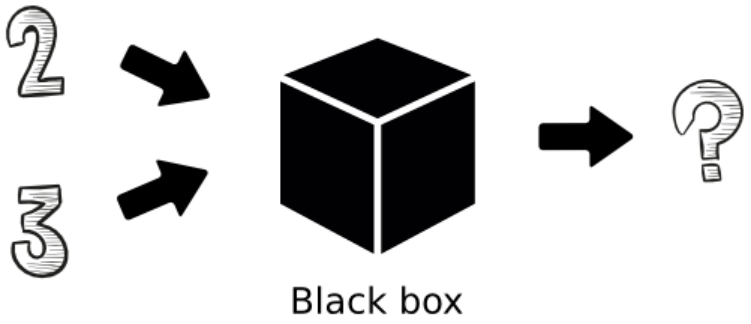


Figure 1.3 An ML approach to solving problems can be thought of as tuning the parameters of a black box until it produces satisfactory results.

PARAMETERS

Sometimes the best way to devise an algorithm that transforms an input to its corresponding output is too complicated. For example, if the input were a series of numbers representing a grayscale image, you can imagine the difficulty in writing an algorithm to label every object seen in the image. Machine learning comes in handy when the inner workings are not well understood. It provides us with a toolset to write software without adequately defining every detail of the algorithm. The programmer can leave some values undecided and let the machine learning system figure out the best values by itself.

The undecided values are called *parameters*, and the description is referred to as the *model*. Your job is to write an algorithm that observes existing examples to figure out how to best tune parameters to achieve the best model. Wow, that's a mouthful! Don't worry, this concept will be a reoccurring motif.

Machine learning might solve a problem without much insight

By mastering this art of inductive problem solving, we wield a double-edged sword. Although ML algorithms may appear to answer correctly to our tests, tracing the steps of deduction to reason why a result is produced may not be as immediate. An elaborate machine learning system learns thousands of parameters, but untangling the meaning behind each parameter is sometimes not the prime directive. With that in mind, I assure you there's a world of magic to unfold.

EXERCISE Suppose you've collected three months-worth of stock market prices. You would like to predict future trends to outsmart the system for monetary gains. Without using ML, how would you go about solving this problem? (As we'll see in chapter 8, this problem becomes approachable using ML techniques.)

LEARNING AND INFERENCE

Suppose you're trying to bake some desserts in the oven. If you're new to the kitchen, it can take days to come up with both the right combination and perfect ratio of ingredients to make something that tastes great. By recording recipes, you can remember how to quickly repeat the dessert if you happen to discover the ultimate tasting meal.

Similarly, machine learning shares this idea of recipes. Typically, we examine an algorithm in two stages: *learning* and *inference*. The objective of the learning stage is to describe the data, which is referred to as the *learned model*. This is our recipe.

Similar to how recipes can be shared and used by other people, the learned model is also reused by other software. The learning stage is the most time-consuming. Running an algorithm may take hours, if not days or weeks, to converge into a useful model. Figure 1.4 outlines the learning pipeline.

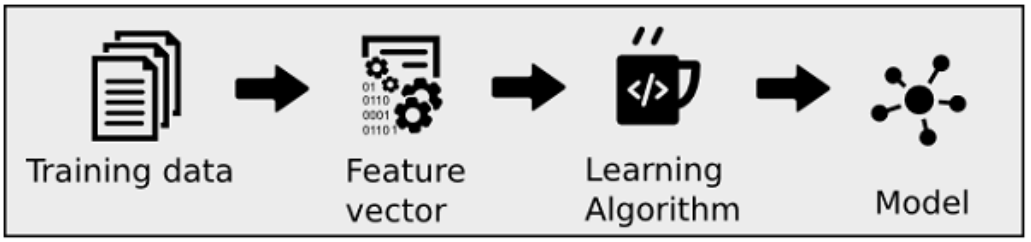


Figure 1.4 The general learning paradigm follows a structured recipe. First, the dataset needs to be transformed into a representation, most often a list of vectors, which can be used by the learning algorithm. The learning algorithm choses a model and efficiently searches for the model's parameters.

The inference stage uses the model to make intelligent remarks about never-before-seen data. It's like using a recipe you found online. The process typically takes orders of magnitude less time than learning, sometimes even being real-time. Inference is all about testing the model on new data, and observing performance in the process, as shown in figure 1.5.

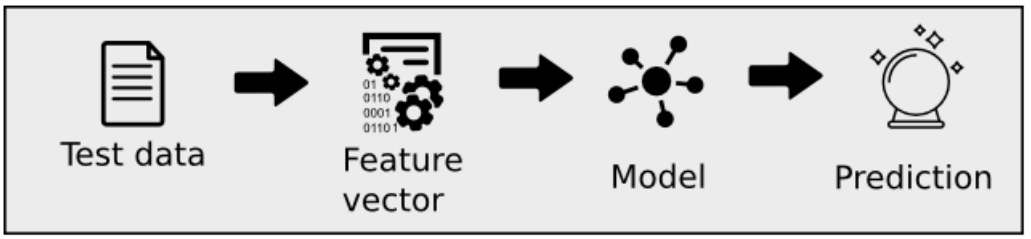


Figure 1.5 The general inference paradigm uses a model that has already been either learned or simply given. After converting data to a usable representation, such as a feature vector, it uses the model to produce intended output.

1.2 Data representation and features

Data is the first class citizen of machine learning. Computers are nothing more than sophisticated calculators, and so the data we feed our machine learning systems must be mathematical objects, such as (1) vectors, (2) matrices, or (3) graphs.

1. *Vectors* have a flat and simple structure and are the typical embodiment of data in most real-world machine learning applications. They have two attributes: a natural number representing the *dimension* of the vector, and a *type* (such as real numbers, integers, and so on). Just as a refresher, some examples of 2-dimension vectors of integers are (1,2) or (-6,0). Some examples of 3-dimension vectors of real numbers are (1.1, 2.0, 3.9) or (π, π/2, π/3). You get the idea, just a collection of numbers of the same type.
2. Moreover, a vector of vectors is a *matrix*.
3. *Graphs*, on the other hand, are more expressive. A graph is a collection of objects (i.e. *nodes*) that can be linked together with *edges* to represent a network. A graphical structure enables representing relationships between objects, such as in a friendship network or a navigation route of a subway system. Consequently, they are tremendously harder to manage in machine learning applications. In this book, our input data will rarely involve a graphical structure.

Either way, a common theme in all forms of representation is the idea of *features*, which are observable properties of an object. The following scenario will help explain this further. When you're in the market for a new car, keeping tabs on every minor detail between different makes and models is essential. After all, if you're about to spend thousands of dollars, you may as well do so diligently. You would likely record a list of features about each car and compare them back and forth. An ordered list of features is often called a *feature vector*.

When buying cars, comparing mileage might be more lucrative than comparing something less relevant to your interest, such as weight. The number of features to track also must be just right, not too few and not too many. This tremendous effort to select both the number of measurements and which measurements to compare is called *feature engineering*. Depending

on which features you examine, the performance of your system can fluctuate dramatically. Selecting the right features to track can make up for a weak learning algorithm.

The general rule of thumb in ML is that more data produces better results. However, the same is not always true for having more features. Perhaps counterintuitive, if the number of features you're tracking is too high, then it may hurt performance. Scholars call this phenomenon the *curse of dimensionality*. Populating the space of all data with representative samples requires exponentially more data as the dimension of the feature vector increases. As a result, feature engineering is one of the most important problems in ML.

Figure 1.6 Feature engineering is the process of selecting relevant features for the task.

You may not appreciate it immediately, but something consequential happens when you decide which features are worth observing. For centuries, philosophers have pondered the meaning of *identity*; you may not immediately realize this, but you've come up with a definition of identity by your choice of specific features.

Imagine writing a machine learning system to detect faces in an image. Let's say one of the necessary features for something to be a face is the presence of two eyes. Implicitly, a face is now defined as something with eyes. Do you realize what types of trouble this can get us into? If a photo of a person shows him or her blinking, then our detector will not find a face because it couldn't find two eyes. The algorithm would fail to detect a face when a person in blinking. The definition of a face was inaccurate to begin with, and it's apparent from the poor detection results.

The identity of an object is decomposed into the features from which it's composed. For example, if the features you are tracking of one car exactly match the corresponding features of another car, they may as well be indistinguishable from your perspective. When hand-

crafting features, we must take great care not to fall into this philosophical predicament of identity.

EXERCISE Let's say you're teaching a robot how to fold clothes. The perception system sees a shirt lying on a table (figure 1.7). You would like to represent the shirt as a vector of features so you can compare it with different clothes. Decide which features would be most useful to track.

Figure 1.7 A robot is trying to fold a shirt. What are good features of the shirt to track?

EXERCISE Now, instead of detecting clothes, you ambitiously decide to detect arbitrary objects. What are some salient features that can easily differentiate objects?

Figure 1.8 Here are images of three objects: a lamp, a pair of pants, and a dog. What are some good features that you should record to compare and differentiate objects?

Feature engineering is a refreshingly philosophical pursuit. For those who enjoy thought-provoking escapades into the meaning of self, I invite you to meditate on feature selection, as it is still an open problem. Fortunately for the rest of you, to alleviate extensive debates, recent advances have made it possible to automatically determine which features to track. You will be able to try it out for yourself in the chapter 8 about autoencoders.

Feature vectors are used in both learning and inference

The interplay between learning and inference is the complete picture of a machine learning system, as seen in figure 1.9. The first step is to represent real-world data into a feature vector. For example, we can represent images by a vector of numbers corresponding to pixel intensities (We'll explore how to represent images in greater detail in future chapters). We can show our learning algorithm the ground truth labels (such as "Bird" or "Dog") along with each feature vector. With enough data, the algorithm generates a learned model. We can use this model on other real-world data to uncover previously unknown labels.

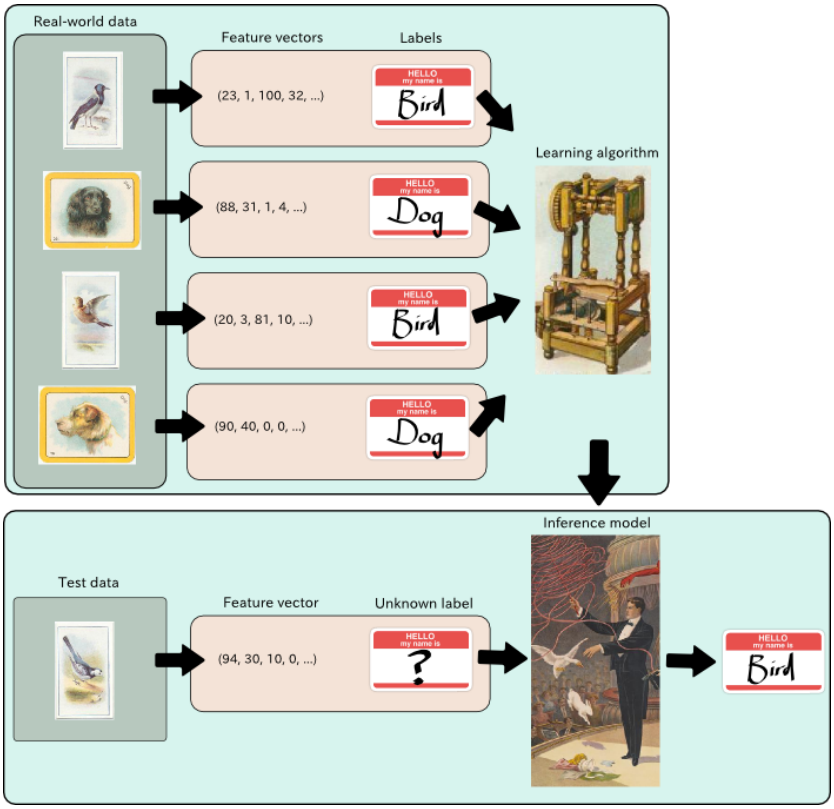


Figure 1.9 Feature vectors are a representation of real world data used by both the learning and inference components of machine learning. The input to the algorithm is not the real-world image directly, but instead its feature vector.

1.3 Distance Metrics

If you have feature vectors of potential cars you want to buy, you can figure out which two are most similar by defining a distance function on the feature vectors. Comparing similarities between objects is an essential component of machine learning. Feature vectors allow us to represent objects so that they we may compare them in a variety of ways. A standard approach is to use the *Euclidian distance*, which is the geometric interpretation you may find most intuitive when thinking about points in space.

Let's say we have two feature vectors, $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$. The Euclidian distance $||x-y||$ is calculated by

$$(\Sigma(x_n - y_n)^2)^{1/2}$$

For example, the Euclidian distance between (0, 1) and (1, 0) is

$$\begin{aligned} &||(0, 1) - (1, 0)|| \\ =\ &||(-1, 1)|| \\ =\ &((-1)^2 + 1^2)^{1/2} \\ =\ &\sqrt{2} \approx 1.414. \end{aligned}$$

Scholars call this the *L2 norm*. But that's actually just one of many possible distance functions. There also exists L0, L1, and L-infinity norms. All of these norms are a valid way to measure distance. Here they are in more detail:

- The *L0 norm* counts the total number of non-zero elements of a vector. For example, the distance between the origin (0, 0) and vector (0, 5) is 1, because there is only 1 non-zero element.
- The *L1 norm* is defined as $\Sigma|x_n|$. The distance between two vectors under the L1 norm is also referred to as the *Manhattan distance*. Imagine living in a downtown area like Manhattan, New York, where the streets form a grid. The shortest distance from one intersection to another is along the blocks. Similarity, the L1 distance between two vectors is along the orthogonal directions. So the distance between (0, 1) and (1, 0) under the L1 norm is 2.

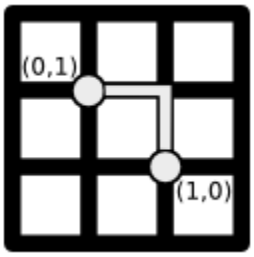


Figure 1.10 The L1 distance is also called the taxi-cab distance because it resembles the route of a car in a grid-like neighborhood such as Manhattan. If a car is travelling from point (0,1) to point (1,0), the shortest route requires a length of 2 units.

- The *L2 norm* is the Euclidian length of a vector, $(\Sigma(x_n)^2)^{1/2}$. It is the most direct route one can possibly take on a geometric plane to get from one point to another.

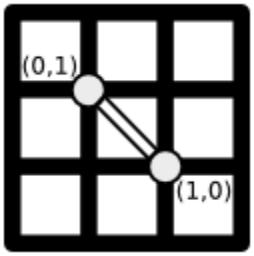


Figure 1.11 The L2 norm between points (0,1) and (1,0) is the length of a single straight line segment reaching both points.

- The *L-N norm* generalizes this pattern, resulting in $(\Sigma|x_n|^N)^{1/N}$. We rarely use finite norms above L2, but it's here for completeness.
- The *L-infinity norm* is $(\Sigma|x_n|^\infty)^{1/\infty}$. More naturally, it is the largest magnitude among each element. If the vector is (-1, -2, -3), t the L-infinity norm will be 3.

When do I use a metric other than the L2 norm in the real-world?

Let's say you're working for a new search-engine start-up trying to compete with established tech giants in the field. Your boss assigns you the task of using machine learning to personalize the search results for each user.

In addition, your boss wants a guarantee that users can only receive less than 5 erroneous search-results per month. Given a vector corresponding to the number of incorrect results shown per user in a month, you are trying to satisfy the boss' condition that the L-infinity norm must be below 5.

Suppose instead that your boss changes his/her mind and requires that less than 5 erroneous search-results can occur for all users collectively. In this case, you are trying to achieve a L1 norm below 5.

Actually, your boss changes his/her mind again. Now, the number of people with erroneous search-results should be less than 5. In that case, you are trying to achieve an L0 norm below 5.

Now that we can compare feature vectors, we have the tools necessary to use data for practical algorithms. Machine learning is often split into three perspectives: supervised learning, unsupervised learning, and reinforcement learning. Let's examine each, one by one.

1.4 Supervised Learning

By definition, a supervisor is someone higher up in the chain of command. When in doubt, he or she dictates what to do. Likewise, *supervised learning* is all about learning from examples laid out by a "supervisor" (such as a teacher).

A supervised machine learning system needs labeled data to develop a useful understanding, which we call its model. For example, given many photographs of people and

their recorded corresponding ethnicity, we can train a model to classify the ethnicity of a never-before-seen individual in an arbitrary photograph. Simply put, a model is a function that assigns a label to some data. It does so by using previous examples, called a *training dataset*, as reference.

A convenient way to talk about models is through mathematical notation. Let x be an instance of data, such as a feature vector. The corresponding label associated with x is $f(x)$, often referred to as the *ground truth* of x. Usually, we use the variable $y = f(x)$ because it's quicker to write. In the example of classifying the ethnicity of a person through a photograph, x can be a hundred-dimensional vector of various relevant features, and y is one of a couple values to represent the various ethnicities. Since y is discrete with few values, the model is called a *classifier*. If y could result in many values, and the values have a natural ordering, then the model is called a *regressor*.

Let's denote a model's prediction of x as $g(x)$. Sometimes you can tweak a model to change its performance drastically. Models have some parameters that can be tuned either by a human or automatically. We use the vector θ to represent the parameters. Putting it all together, $g(x|\theta)$ more completely represents the model, read "g of x given θ."

> **ASIDE** Models may also have *hyper-parameters*, which are extra ad-hoc properties about a model. The word "hyper" in "hyper-parameter" is a bit strange at first. If it helps, a better name could be "meta-parameter," because the parameter is akin to metadata about the model.

The success of a model's prediction $g(x|\theta)$ depends on how well it agrees with the ground truth y. We need a way to measure the distance between these two vectors. For example, the L2-norm may be used to measure how close two vectors lie. The distance between the ground truth and prediction is called the *cost*.

The essence of a supervised machine learning algorithm is to figure out the parameters of a model that results in the least cost. Mathematically put, we are trying to look for a θ^* that minimizes the cost among all data points $x \in X$.

$$\theta^* = \arg\min \text{Cost}(\theta \mid X)$$
$$\text{where } Cost(\theta \mid X) = \sum_x dist(\, g(x \mid \theta) - f(x)\,)$$
$$\text{and } dist \text{ is a distance metric such as the L2-norm}$$

Clearly, brute forcing every possible combination of θs, also known as a *parameter-space*, will eventually find the optimal solution, but at an unacceptable runtime. A major study in machine learning is about writing algorithms that efficiently search through this parameter-space. Some of the first algorithms include *gradient descent*, *simulated annealing*, and *genetic algorithms*. TensorFlow automatically takes care of the low-level implementation details of these algorithms, so we won't get into them in too much detail.

Once the parameters are learned one way or another, you can finally evaluate the model to figure out how well the system captured patterns from the data. A rule of thumb is not to evaluate your model on the same data you used to train it. Use the majority of the data for

training, and the remaining for testing. For example, if you have 100 labeled data, randomly select 70 of them to train a model, and reserve the other 30 to test it.

Why split the data?

If the 70-30 split seems odd to you, think about it like this. Let's say your Physics teacher gives you a practice exam and tells you the real exam will be no different. You might as well memorize the answers and earn a perfect score without actually understanding the concepts. Similarly, if we test our model on the training dataset, we're not doing ourselves any favors. We risk a false sense of security since the model may merely be memorizing the results. Now, where's the intelligence in that?

Actually, instead of the 70-30 split, machine learning practitioners typically divided their dataset 60-20-20. Training consumes 60% of the dataset, and testing uses 20%, leaving the other 20% for what is called "validation," which will be explained in the next chapter.

1.5 Unsupervised Learning

Unsupervised learning is about modeling data that comes without corresponding labels or responses. The fact that we can make any conclusions at all on just raw data feels like magic. With enough data, it may be possible to find patterns and structure. Two of the most powerful tools that machine learning practitioners use to learn from data alone are *clustering* and *dimensionality reduction*.

Clustering is the process of splitting the data into individual buckets of similar items. In a sense, clustering is like classification of data without knowing any corresponding labels. For instance, when reorganizing a bookshelf, you likely place similar genres together, or maybe you group them by author's last name. One of the most popular clustering algorithms is *K-means*, which is a specific instance of a more powerful technique called the *E-M algorithm*.

Dimensionality reduction is about manipulating the data to view it under a much simpler perspective. It is the ML equivalent of the phrase, "Keep it simple, stupid!" For example, by getting rid of redundant features, we can explain the same data in a lower-dimensional space and see which features really matter. This simplification also helps in data visualization or preprocessing for performance efficiency. One of the earliest algorithms is Principle Component Analysis (PCA), and some newer ones include autoencoders, which we'll cover in chapter 7.

1.6 Reinforcement Learning

Supervised and unsupervised learning seem to suggest that the existence of a teacher is all or nothing. But, there is a well-studied branch of machine learning where the environment acts as a teacher, providing hints as opposed to definite answers. The learning system receives feedback on its actions, with no concrete promise that it's progressing in the right direction.

Exploration vs. Exploitation is the heart of reinforcement learning

Imagine playing a video-game that you've never seen before. You click buttons on a controller and discover that a particular combination of strokes gradually increases your score. Brilliant, now you repeatedly exploit this finding in hopes of beating the high-score. In the back of your mind, you think to yourself that maybe there's a better combination of button-clicks that you're missing out on. Should you exploit your current best strategy, or risk exploring new options?

Unlike supervised learning, where training data is conveniently labeled by a "teacher," *reinforcement learning* trains on information gathered by observing how the environment reacts to actions. In other words, reinforcement learning is a type of machine learning that interacts with the environment to learn which combination of actions yields the most favorable results. Since we're already anthropomorphizing our algorithm by using the words "environment" and "action," scholars typically refer to the system as an autonomous "agent." Therefore, this type of machine learning naturally manifests itself into the domain of robotics.

To reason about agents in the environment, we introduce two new concepts: states and actions. The status of the world frozen at some particular time is called a *state*. An agent may perform one of many *actions* to change the current state. To drive an agent to perform actions, each state yields a corresponding *reward*. An agent eventually discovers the expected total reward of each state, called the *value* of a state.

Like any other machine learning system, performance improves with more data. In this case, the data is a history of previous experiences. In reinforcement learning, we do not know the final cost or reward of a series of actions until it's executed. These situations render traditional supervised learning ineffective. The only information an agent knows for certain is the cost of a series of actions that it has already taken, which is incomplete. The agent's goal is to find a sequence of actions that maximizes rewards.

EXERCISE Would you use supervised, unsupervised, or reinforcement learning to solve the following problems? (a) Organize various fruits in 3 baskets based on no other information. (b) Predict the weather based off sensor data. (c) Learn to play chess well after many trial-and-error attempts.

ANSWER (a) unsupervised, (b) supervised, (c) reinforcement

1.7 Existing Tools

One of the easiest ways to get started with machine learning and data analysis is through the *Scikit-learn* Python library (http://scikit-learn.org/). Python is a great language to prototype ideas that eventually become industry standard implementations. Some of the most enduring and successful Python libraries such as NumPy, SciPy, and matplotlib form the backbone of Scikit-learn. The tools are simple, making them easy to use.

However, Scikit-learn feels like assembly language because the library is relatively low-level. As a result, performing sophisticated algorithms can easily result in buggy code. Instead

of interacting directly with Scikit-learn, using higher-level libraries such as TensorFlow, Theano, or Caffe offers a more robust setup while sacrificing some flexibility.

Hadoop or Apache Spark are some higher-level industry standard frameworks to deal with big data where the emphasis is parallelism and distributed computing. Commonly, the lower-level library used to interact with these parallel architectures is Apache *Mahout*, providing a complete Java interface.

Apart from TensorFlow, some of the most common machine learning libraries include Theano, Caffe, Torch, and Computational Graph Toolkit as shown in Figure 1.12.

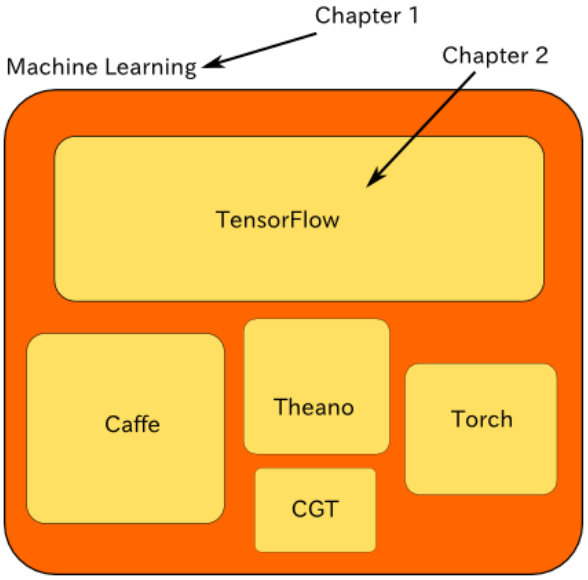


Figure 1.12 TensorFlow is one the most popular machine learning libraries. At the time of writing, it has more than twice as many favorites on GitHub than Caffe, the next most popular library. The size of the rectangles roughly corresponds to popularity on GitHub.

TensorFlow was released to the public in November 2015. Before that, the following were some of the most commonly used machine learning libraries:

1.7.1 Theano

Throughout the years, machine learning practitioners have already implemented many well-known neural networks in Theano. It uses a symbolic graph to decouple implementation from design. The programming environment is in Python, which makes it easy for a novice to jump in and give it a go regardless of platform. There is little support, however, for a low-level interface to make substantial customization. Moreover, there is a considerable overhead for

rapid prototyping since it compiles code to binary every time, making a quick modification or debugging a burden.

1.7.2 Caffe

Caffe's primary interactions are easy to use because of its simple Python interface. For more complex algorithms or customized neural networks, one can also use the C++ interface. Since most platforms support C++, it is readily deployable. However, the purpose of Caffe is primarily for networks that deal with images. Text or time-series data will be unnatural to process though Caffe.

1.7.3 Torch

Lua is the programming language of choice for this framework. Since the Lua environment is foreign to most developers, there's a nontrivial risk associated with using Torch for machine learning projects. But on the bright side, Torch has strong support for optimization solvers so that one wouldn't need to reinvent the wheel.

1.7.4 Computational Graph Toolkit

A lab in University of California, Berkeley released Computational Graph Toolkit (CGT) for generalized graph operations, often used in machine learning. It supports some of the same features as Theano, but with an emphasis on parallelism. Unfortunately, documentation is relatively sparse compared to the other libraries, and the community is not as prevalent as that of Theano or Caffe.

1.8 TensorFlow

Google open-sourced their machine learning framework called TensorFlow in late 2015 under the Apache 2.0 license. Before that, it was used proprietarily by Google in its speech recognition, Search, Photos, and Gmail, among other applications.

A bit of history

A former scalable distributed training and learning system called DistBelief is the primary influence on TensorFlow's current implementation. Ever written a messy piece of code and wished you could start all over again? That's essentially the dynamic between DistBelief and TensorFlow.

The library is implemented in C++ and has a convenient Python API, as well a lesser appreciated C++ API. As a result of the simpler dependencies, TensorFlow can be quickly deployed to various architectures.

Similar to Theano, computations are described as flowcharts, separating design from implementation. With little to no hassle, this dichotomy allows the same design to be

implemented on not just large-scale training systems with thousands of GPUs, but also simply on mobile devices. The single system spans a broad range of platforms.

One of the fanciest properties of TensorFlow is its *automatic differentiation* capabilities. One can experiment with new networks without having to redefine many key calculations.

> **ASIDE** Automatic differentiation makes it much easier to implement backpropagation, which is a computationally heavy calculation used in a branch of machine learning called neural networks. TensorFlow hides the nitty-gritty details of backpropagation so that you can focus on the bigger picture. Chapter 7 covers an introduction to neural networks with TensorFlow.

All the mathematics is abstracted away and unfolded under the hood. It's like using WolframAlpha for a Calculus problem set.

Another feature of this library is its interactive visualization environment called TensorBoard. This tool shows a flowchart of how data transforms, displays summary logs over time, as well as traces performance. Figure 1.13 shows an example of what TensorBoard looks like when in use. The next chapter will cover using it in greater detail.

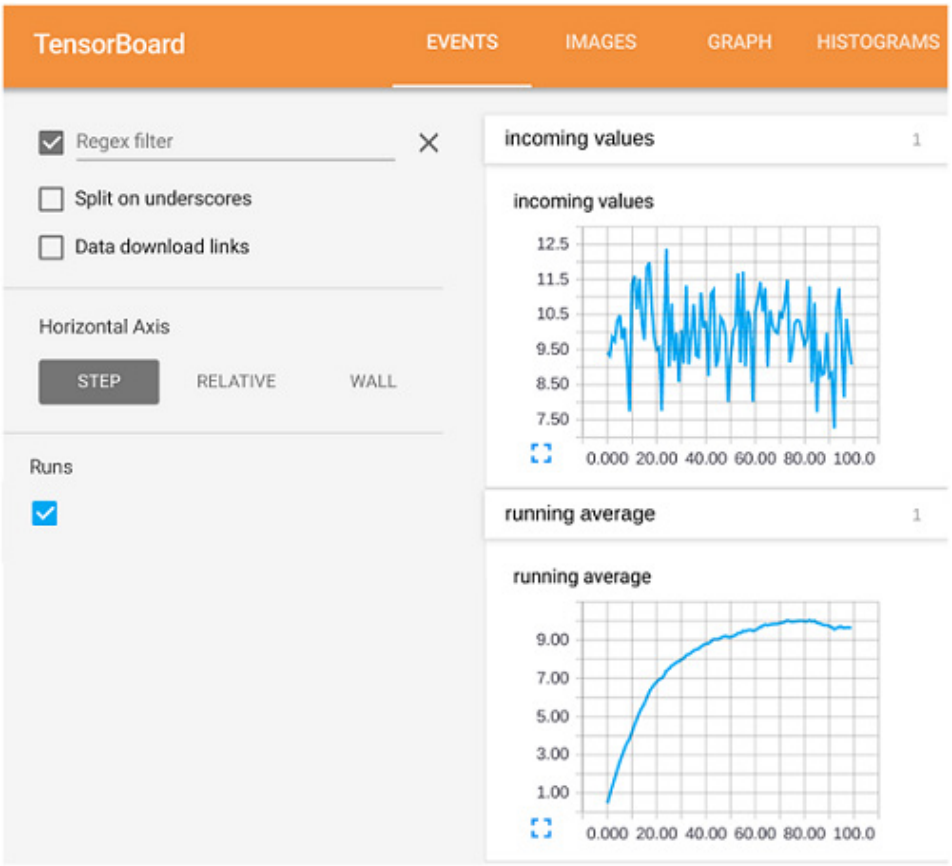


Figure 1.13 Example of TensorBoard in action

Unlike Theano, prototyping in TensorFlow is much faster (code initiates in a matter of seconds as opposed to minutes) because much of the operations come pre-compiled. It becomes easy to debug code due to subgraph execution. What that means is that an entire segment of computation can be reused without recalculation.

Because TensorFlow is not only about neural networks, it also has out-of-the-box matrix computation and manipulation tools. Most libraries such as Torch or Caffe are designed solely for deep neural networks, but TensorFlow is more flexible.

The library is well documented and is officially supported by Google. Machine learning is a sophisticated topic, so having an exceptionally well-reputed company behind TensorFlow is comforting.

1.9 Overview of future chapters

Chapters 3-6 are about how to implement classic machine learning algorithms in TensorFlow, whereas chapters 7-11 cover algorithms based on neural networks (see figure 1.14). The algorithms solve a wide variety of problems such as prediction, classification, clustering, dimensionality reduction, and planning.

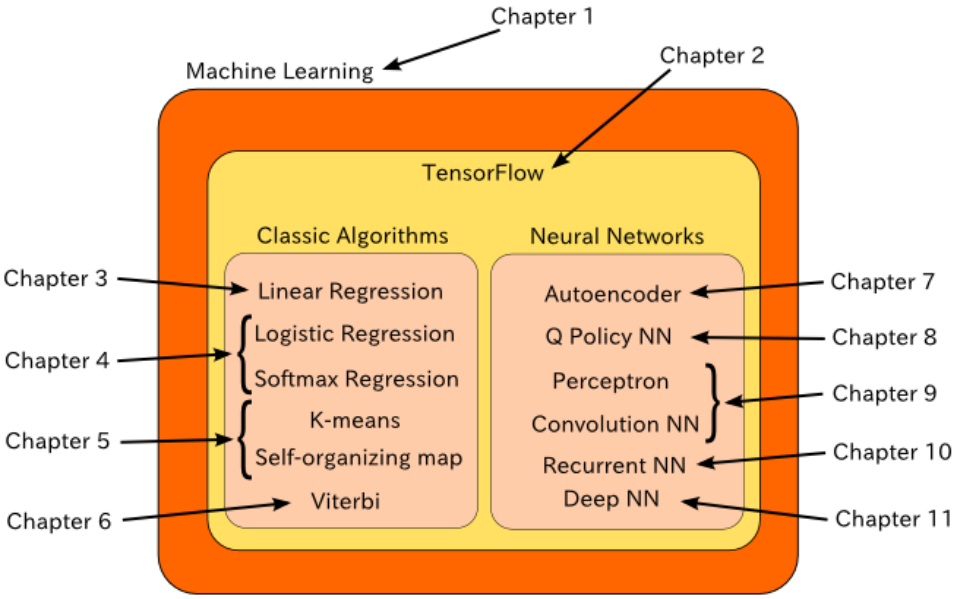


Figure 1.14 The technical chapters are divided into two categories of algorithms: (1) classic algorithms, and (2) neural networks.

There are many algorithms to solve the same real world problem, and many real world problems that are solved by the same algorithm, but table 1.1 covers the ones laid out in this book.

Table 1.1 Many real-world problems can be solved using the corresponding algorithm found in their respective chapters.

Real world problem	Algorithm	Chapter
Predicting trends, fitting a curve to data points, describing relationships between variables	Linear regression	3
Classifying data into two categories, finding the best way to split a dataset	Logistic regression	4
Classifying data into multiple categories	Softmax regression	4

Revealing hidden causes of observations, finding the most likely hidden reason for a series of outcomes	Hidden Markov Model (Viterbi)	5
Clustering data into a fixed number of categories, automatically partitioning data points into separate classes	K-means	6
Clustering data into arbitrary categories, visualizing high-dimensional data into a lower-dimensional embedding	Self-organizing map	6
Reducing dimensionality of data, learning latent variables responsible for high-dimensional data	Autoencoder	7
Planning actions in an environment using neural networks (reinforcement learning)	Q Policy neural network	8
Classifying data using supervised neural networks	Perceptron	9
Classifying real-world images using supervised neural networks	Convolution neural network	9
Producing patterns that match observations using neural networks	Recurrent neural network	10

1.10 Summary

You learned quite a bit about machine learning in this chapter, including the following:

- Machine learning is about using examples to develop an expert system that can make useful statements about new inputs.
- A key property of ML is that performance tends to improve with more training data.
- Over the years, scholar have crafted three major archetypes that most problems fit: supervised learning, unsupervised learning, and reinforcement learning.
- After a real-world problem is formulated in a machine learning perspective, a number of algorithms become available. Out of the many software libraries and frameworks to accomplish an implementation, we chose TensorFlow as our silver bullet. Developed by Google and supported by its flourishing community, TensorFlow gives us a way to easily implement industry standard code.

In the next chapter, we'll get our hands dirty with TensorFlow to get comfortable with using the library.

2

TensorFlow essentials

This chapter covers

- The TensorFlow workflow
- Creating interactive notebooks with Jupyter
- Visualizing algorithms using TensorBoard

Before implementing machine learning algorithms, let's first familiarize ourselves with how to use TensorFlow. You're going to get your hands dirty writing some simple code right away! This chapter will cover some essential advantages of TensorFlow to convince you it's the machine learning library of choice.

As a thought experiment, let's see what happens when we use Python code without a handy computing library. It'll be like using a new smartphone without installing any additional apps. The functionality will be there, but you'd be so much more productive if you had the right tools.

Suppose you're a private business owner tracking the flow of sales for your products. Your inventory consists of 100 different items, and you represent each item's price in a vector called `prices`. Another 100-dimensional vector called `amounts` represents the inventory count of each item. You can write the following chunk of Python code shown in listing 2.1 to calculate the revenue of selling all products. Keep in mind that this code does not import any libraries.

Listing 2.1 Computing the inner product of two vectors without using a library

```
revenue = 0
for price, amount in zip(prices, amounts):
    revenue += price * amount
```

That's a lot of code just to calculate the inner-product of two vectors (also known as *dot product*). Imagine how much code would be required for something more complicated, such as solving linear equations or computing the distance between two vectors.

By installing the TensorFlow library, you also end up installing a well-known and robust Python library called NumPy, which facilitates mathematical manipulation in Python. Using Python without its libraries (NumPy and TensorFlow) is like using a camera without autofocus: you gain more flexibility, but you can easily make careless mistakes (for the record, I have nothing against photographers who micro-manage aperture, shutter, and ISO). It's easy to make mistakes in machine learning, so let's keep our camera on auto-focus and use TensorFlow to help automate some tedious software development.

Listing 2.2 shows how to concisely write the same inner-product using NumPy.

Listing 2.2 Computing the inner product using NumPy

```
import numpy as np
revenue = np.dot(prices, amounts)
```

Python is a succinct language. Fortunately for you, that means this book will not have pages and pages of cryptic code. On the other hand, the brevity of the Python language also implies that a lot is happening behind each line of code, which you should familiarize yourself with carefully as you follow along the chapter.

Machine learning algorithms require a large amount of mathematical operations. Often an algorithm boils down to a composition of simple functions iterated until convergence. Sure, you may use any standard programming language to perform these computations, but the secret to both manageable and performant code is the use of a well-written library, such as TensorFlow (which officially supports Python and C++).

Official TensorFlow library reference

Detailed documentation about various functions for the Python and C++ APIs are available at https://www.tensorflow.org/versions/r0.8/api_docs/index.html.

The skills you learn in this chapter are geared toward using TensorFlow for computations, because machine learning relies on mathematical formulations. After going through the examples and code listings, you will be able to use TensorFlow for arbitrary tasks, such as computing statistics on big data. The focus here will entirely be about how to use TensorFlow, as opposed to machine learning. That sounds like a gentle start, right?

Later on in this chapter, we'll use TensorFlow's flagship features that are essential for machine learning. These include representation of computation as a data-flow graph, separation of design and execution, partial subgraph computation, and auto-differentiation. Without further ado, let's write our first TensorFlow code!

2.1 Ensuring TensorFlow works

First of all, you should ensure everything is working correctly. Check the oil level in your car, repair the blown fuse in your basement, and ensure that your credit balance is zero.

Just kidding, I'm talking about TensorFlow.

Before you begin, follow the procedures in appendix A for step-by-step installation instructions. Create a new file called `test.py` for our first piece of code. Import TensorFlow by running the following script:

```
import tensorflow as tf
```

This single import prepares TensorFlow for your bidding. If the Python interpreter doesn't complain, then we're ready to start using TensorFlow!

Having technical difficulty?

A common cause of error at this step is if you installed the GPU version and the library fails to search for CUDA drivers. Remember, if you compiled the library with CUDA, you need to update your environment variables with the path to CUDA. Check the CUDA instructions on TensorFlow. (See https://www.tensorflow.org/versions/master/get_started/os_setup.html#optional-linux-enable-gpu-support for further information).

Sticking with TensorFlow conventions

The TensorFlow library is usually imported with the `tf` qualified name. Generally, qualifying TensorFlow with `tf` is a good idea to remain consistent with other developers and open-source TensorFlow projects. Of course, you may choose not to qualify it or change the qualification name, but then successfully reusing other people's snippets of TensorFlow code in your own projects will be an involved process.

2.2 Representing tensors

Now that we know how to import TensorFlow into a Python source file, let's start using it! As covered in the previous chapter, a convenient way to describe an object in the real world is through listing out its properties, or features. For example, you can describe a car by its color, model, engine type, mileage, and so on. An ordered list of some features is called a *feature vector,* and that's exactly what we'll represent in TensorFlow code.

Feature vectors are one of the most useful devices in machine learning because of their simplicity (they're just a list of numbers). Each data item typically consists of a feature vector, and a good dataset has hundreds, if not thousands, of these feature vectors. No doubt, you'll often be dealing with more than one vector at a time. A *matrix* concisely represents a list of vectors, where each column of a matrix is a feature vector.

The syntax to represent matrices in TensorFlow is a vector of vectors, each of the same length. Figure 2.1 is an example of a matrix with two rows and three columns, such as `[[1, 2, 3], [4, 5, 6]]`. Notice, this is a vector containing two elements, and each element corresponds to a row of the matrix.

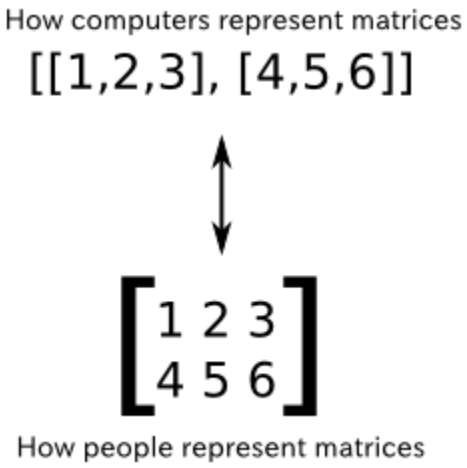


Figure 2.1 The matrix in the lower half of the diagram is a visualization from its compact code notation in the upper half of the diagram. This form of notation is a common paradigm in most scientific computing libraries.

We access an element in a matrix by specifying its row and column indices. For example, the first row and first column indicate the very first top-left element. Sometimes it's convenient to use more than two indices, such as when referencing a pixel in a color image not only by its row and column, but also its red/green/blue channel. A *tensor* is a generalization of a matrix that specifies an element by an arbitrary number of indices.

Example of a tensor

Suppose an elementary school enforces assigned seating to all its students. You're the principal, and you're terrible with names. Luckily, each classroom has a grid of seats, where you can easily nickname a student by his or her row and column index.

There are multiple classrooms, so you cannot simply say "Good morning 4,10! Keep up the good work." You need to also specify the classroom, "Hi 4,10 from classroom 2." Unlike a matrix, which needs only two indices to specify an element, the students in this school need three numbers. They're all a part of a rank three tensor!

The syntax for tensors is even more nested vectors. For example, a 2-by-3-by-2 tensor is `[[[1,2], [3,4], [5,6]], [[7,8], [9,10], [11,12]]]`, which can be thought of as two matrices, each of size 3-by-2. Consequently, we say this tensor has a *rank* of 3. In general, the rank of a tensor is the number of indices required to specify an element. Machine learning algorithms in TensorFlow act on Tensors, so it's important to really understand how to use them.

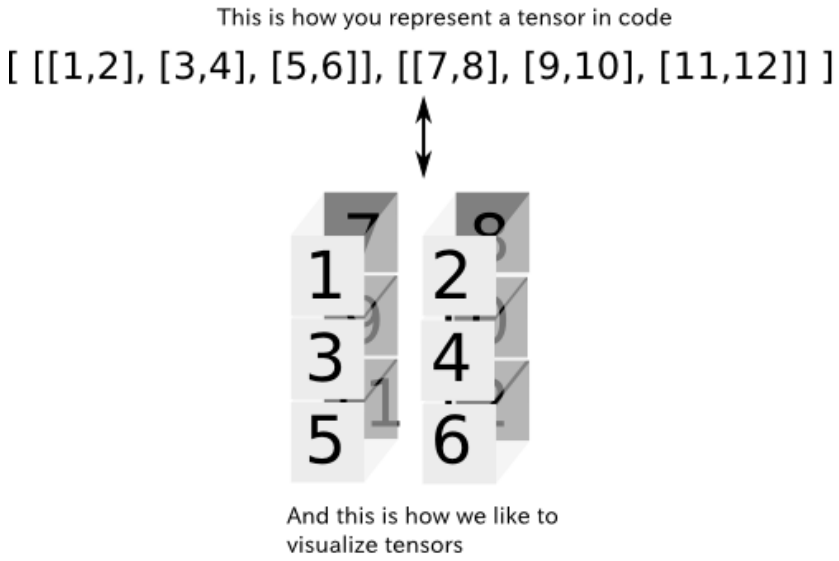


Figure 2.2 This tensor can be thought of as multiple matrices stacked on top of each other. To specify an element, you must indicate the row and column, as well as which matrix is being accessed. Therefore, the rank of this tensor is 3.

It's easy to get lost in the many ways to represent a tensor. Intuitively, each of the following three lines of code in Listing 2.3 is trying to represent the same 2-by-2 matrix. This matrix represents two features vectors of two dimensions each. It could, for example, represent two people's ratings of two movies. Each person, indexed by the row of the matrix, assigns a number to describe his or her review of the movie, indexed by the column. Run the code to see how to generate a matrix in TensorFlow.

Listing 2.3 Different ways to represent tensors

```
import tensorflow as tf
import numpy as np    ❶

m1 = [[1.0, 2.0],
      [3.0, 4.0]]     ❷
m2 = np.array([[1.0, 2.0],
               [3.0, 4.0]], dtype=np.float32)    ❷
m3 = tf.constant([[1.0, 2.0],
                  [3.0, 4.0]])                   ❷

print(type(m1))       ❸
print(type(m2))       ❸
print(type(m3))       ❸

t1 = tf.convert_to_tensor(m1, dtype=tf.float32) ❹
t2 = tf.convert_to_tensor(m2, dtype=tf.float32) ❹
t3 = tf.convert_to_tensor(m3, dtype=tf.float32  ❹
```

```
print(type(t1))      ❺
print(type(t2))      ❺
print(type(t3))      ❺
```

❶ We'll use NumPy matrices in TensorFlow
❷ Define a 2x2 matrix in 3 different ways
❸ Print the type for each matrix
❹ Create tensor objects out of the various different types
❺ Notice that the types will be the same now

The first variable (`m1`) is a list, the second variable (`m2`) is an `ndarray` from the NumPy library, and the last variable (`m3`) is TensorFlow's `Tensor` object. All operators in TensorFlow, such as `neg`, are designed to operate on tensor objects. A convenient function we can sprinkle anywhere just to make sure that we're dealing with tensors as opposed to the other types is `tf.convert_to_tensor( ... )`. In fact, most functions in the TensorFlow library already perform this function (redundantly) even if you forget to do so. Using `tf.convert_to_tensor( ... )` is optional, but I show it here because it helps demystify the implicit type system being handled across the library. The previous listing 2.3 outputs the following three times:

```
<class 'tensorflow.python.framework.ops.Tensor'>
```

Let's take another look at defining tensors in code. After importing the TensorFlow library, we can use the constant operator as follows in Listing 2.4.

Listing 2.4 Creating tensors

```
import tensorflow as tf

matrix1 = tf.constant([[1., 2.]])        ❶
matrix2 = tf.constant([[1],
                       [2]])             ❷
myTensor = tf.constant([ [[1,2],
                          [3,4],
                          [5,6]],
                         [[7,8],
                          [9,10],
                          [11,12]] ])     ❸

print(matrix1)     ❹
print(matrix2)     ❹
print(myTensor)    ❹
```

❶ Define a 2x1 matrix
❷ Define a 1x2 matrix
❸ Define a rank 3 tensor
❹ Try printing the tensors

Running listing 2.4 produces the following output:

```
Tensor( "Const:0",
        shape=TensorShape([Dimension(1), Dimension(2)]),
        dtype=float32 )
Tensor( "Const_1:0",
        shape=TensorShape([Dimension(2), Dimension(1)]),
        dtype=int32 )
Tensor( "Const_2:0",
        shape=TensorShape([Dimension(2), Dimension(3), Dimension(2)]),
        dtype=int32 )
```

As you can see from the output, each tensor is represented by the aptly named Tensor object. Each Tensor object has a unique label (`name`), a dimension (`shape`) to define its structure, and data type (`dtype`) to specify the kind of values we will manipulate. Because we did not explicitly provide a name, the library automatically generated the names: "`Const:0`", "`Const_1:0`", and "`Const_2:0`".

Tensor types

Notice that each of the elements of `matrix1` end with a decimal point. The decimal point tells Python that the data type of the elements is not an integer, but instead a float. We can pass in explicit `dtype` values. Much like NumPy arrays, tensors take on a data type that specifies the kind of values we'll manipulate in that tensor.

TensorFlow also comes with a few convenient constructors for some simple tensors. For example, `tf.zeros(shape)` creates a tensor with all values initialized at zero of a specific shape. Similarly, `tf.ones(shape)` creates a tensor of a specific shape with all values initialized at one. The `shape` argument is a one-dimensional (1D) tensor of type `int32` (a list of integers) describing the dimensions of the tensor.

EXERCISE 2.1: Initialize a 500-by-500 tensor with all elements equaling 0.5.

ANSWER tf.ones([500,500]) * 0.5

2.3 Creating operators

Now that we have a few starting tensors ready to be used, we can apply more interesting operators such as addition or multiplication. Consider each row of a matrix representing the transaction of money to (positive value) and from (negative value) another person. Negating the matrix is a way to represent the transaction history of the other person's flow of money. Let's just start simple and run the negation op (short for operation) on our `matrix1` tensor from listing 2.4. Negating a matrix turns the positive numbers into negative numbers of the same magnitude, and vice versa.

Negation is one of the simplest operations. As shown in listing 2.5, negation takes only one tensor as input, and produces a tensor with every element negated. Try running the code. If you master how to define negation, it'll provide a stepping stone to generalize that skill to all other TensorFlow operations.

ASIDE *Defining* an operation, such as negation, is different from *running* it. So far, you've *defined* how operations should behave. In section 2.4, you'll *evaluate* (or *run*) them to compute their value.

Listing 2.5 Using the negation operator

```
import tensorflow as tf

x = tf.constant([[1, 2]])    ❶
neg_x = tf.neg(x)    ❷
print(neg_x)    ❷
```

❶ Define an arbitrary tensor
❷ Negate the tensor
❸ Print the object

Listing 2.5 generates the following output:

```
Tensor("Neg:0", shape=(1, 2), dtype=int32)
```

Useful TensorFlow operators

The official documentation carefully lays out all available math ops:
https://www.tensorflow.org/api_docs/Python/math_ops.html.

Some specific examples of commonly used operators include:

`tf.add(x, y)` → Add two tensors of the same type, x + y
`tf.sub(x, y)` → Subtract tensors of the same type, x - y
`tf.mul(x, y)` → Multiply two tensors element-wise
`tf.pow(x, y)` → Take the element-wise power of x to y
`tf.exp(x)` → Equivalent to `pow(e, x)`, where `e` is Euler's number (2.718...)
`tf.sqrt(x)` → Equivalent to `pow(x, 0.5)`
`tf.div(x, y)` → Take the element-wise division of x and y
`tf.truediv(x, y)` → Same as `tf.div`, except casts the arguments as a float
`tf.floordiv(x, y)` → Same as `truediv`, except rounds down the final answer into an integer
`tf.mod(x, y)` → Takes the element-wise remainder from division

EXERCISE 2.2: Use the TensorFlow operators we've learned so far to produce the Gaussian Distribution (also known as Normal Distribution). See Figure 2.3 for a hint. For reference, you can find the probability density of the normal distribution online: https://en.wikipedia.org/wiki/Normal_distribution.

ANSWER Most mathematical expressions such as "*", "-", "+", etc. are just shortcuts for their TensorFlow equivalent for brevity. The Gaussian function includes many operations, so it's cleaner to use some short-hand notations as follows:

```
from math import pi
mean = 1.0
sigma = 0.0
(tf.exp(tf.neg(tf.pow(x - mean, 2.0) /
               (2.0 * tf.pow(sigma, 2.0) ))) *
 (1.0 / (sigma * tf.sqrt(2.0 * pi) )))
```

2.4 Executing operators with sessions

A session is an environment of a software system that describes how the lines of code should run. In TensorFlow, a session sets up how the hardware devices (such as CPU and GPU) talk to each other. That way, you can design your machine learning algorithm without worrying about micro-managing the hardware that it runs on. Of course, you can later configure the session to change its behavior without changing a line of the machine learning code.

To execute an operation and retrieve its calculated value, TensorFlow requires a session. Only a registered session may fill the values of a Tensor object. To do so, you must create a session class using `tf.Session()` and tell it to run an operator (listing 2.6). The result will be a value you can later use for further computations.

Listing 2.6 Using a session

```
import tensorflow as tf

matrix = tf.constant([[1., 2.]])    ❶
neg_matrix = tf.neg(matrix)         ❷

with tf.Session() as sess:          ❸
    result = sess.run(neg_matrix)   ❹
print(result)                       ❺
```

❶ Define an arbitrary matrix
❷ Run the negation operator on it
❸ Start a session to be able to run operations
❹ Tell the session to evaluate negMatrix
❺ Print the resulting matrix

Congratulations! You have just written your first full TensorFlow code. Although all it does is negate a matrix to produce `[[-1, -2]]`, the core overhead and framework are just the same as everything else in TensorFlow.

Code performance seems a bit slow

You may have noticed that running your code took an extra few seconds than expected. It may appear unnatural that TensorFlow takes seconds to simply negate a small matrix. However, there is substantial preprocessing that occurs to optimize the library for larger and more complicated computations.

Every Tensor object has an `eval()` function to evaluate the mathematical operations that defines its value. However, the `eval()` function requires defining a session object for the

library to understand how best to make use of the underlying hardware. Previously, in listing 2.6, we use `sess.run(…)`, which is equivalent to invoking the Tensor's `eval()` function in context of the session.

When running TensorFlow code through an interactive environment (for debugging or presentation purposes), it is often easier to create the session in interactive mode, where the session is implicitly part of any call to `eval()`. That way, the session variable does not need to be passed around throughout the code, making it easier to focus on the relevant parts of the algorithm, as seen in listing 2.7.

Listing 2.7 Using the interactive session mode

```
import tensorflow as tf
sess = tf.InteractiveSession()      (1)

matrix = tf.constant([[1., 2.]])    (2)
negMatrix = tf.neg(matrix)          (2)

result = negMatrix.eval()           (3)
print(result)                       (4)
sess.close()                        (5)
```

(1) Start an interactive session so the sess variable no longer needs to be passed around
(2) Define some arbitrary matrix and negate it
(3) You can now evaluate negMatrix without explicitly specifying a session
(4) Print the negated matrix
(5) Remember to close the session to free up resources

2.4.1 Understanding code as a graph

Consider a doctor predicting the expected weight of a newborn to be 7.5 pounds. You would like to figure out how that differs from the actual measured weight. Being an overly analytical engineer, you design a function to describe the likelihood of all possible weights of the newborn. For example, 8 pounds is more likely than 10 pounds.

You can choose to use the *Gaussian* (otherwise known as *Normal*) probability distribution function. It takes as input a number, and outputs a non-negative number describing the probability of observing the input. This function shows up all the time in machine learning, and is easy to define in TensorFlow. It uses multiplication, division, negation, and a couple other fundamental operators.

Think of every operator as a node in a graph. The edges between operators represent the composition of mathematical functions. Specifically, the `neg` operator we've been studying so far is a node, and the incoming/outgoing edges of this node are how the Tensor transforms. Here's a thought: every operator is a strongly typed function that takes input tensors of a particular dimension and produces output of a particular dimension. Figure 2.3 is an example of how the Gaussian function can be designed using TensorFlow, represented as a graph where operators are nodes and edges are how they interact.

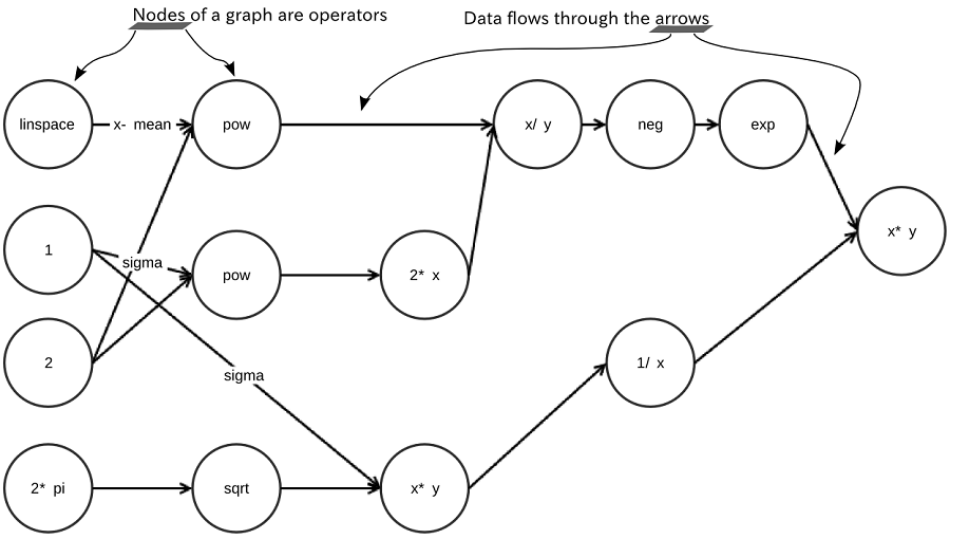


Figure 2.3 The graph represents the operations needed to produce a Gaussian distribution. The links between the nodes represent how data flows from one operation to the next. The operations themselves are very simple, but the complexity arises from how they intertwine.

TensorFlow algorithms are easy to visualize. They can be simply described by flowcharts. The technical (and more correct) term for the flowchart is a *graph*. Every arrow in a flowchart is called the *edge* of the graph. In addition, every state of the flowchart is called a *node*.

2.4.2 Session configurations

You can also pass options to `tf.Session`. For example, TensorFlow automatically determines the best way to assign a GPU or CPU device to an operation, depending on what is available. We can pass an additional option, `log_device_placements=True`, when creating a Session, as shown in listing 2.8.

Listing 2.8 Logging a session

```
import tensorflow as tf

matrix = tf.constant([[1., 2.]])    (1)
negMatrix = tf.neg(matrix)          (1)

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:    (2)
    result = sess.run(negMatrix)    (3)

print(result)                       (4)
```

(1) Define a matrix and negate it
(2) Start the session with a special config passed into the constructor to enable logging
(3) Evaluate negMatrix

❹ **Print the resulting value**

This outputs info about which CPU/GPU devices are used in the session for each operation. For example, running listing 2.8 results in traces of output like the following to show which device was used to run the negation op:

```
Neg: /job:localhost/replica:0/task:0/cpu:0
```

Sessions are essential in TensorFlow code. You need to call a session to actually "run" the math. Figure 2.4 maps out how the different components on TensorFlow interact with the machine learning pipeline. A session not only runs a graph operation, but can also take placeholders, variables, and constants as input. We've used constants so far, but in later sections we'll start using variables and placeholders. Here's a quick overview of these three types of values.

- Placeholder

  A value that is unassigned, but will be initialized by the session wherever it is run.

- Variable

  A value that can change, such as parameters of a machine learning model.

- Constant

  A value that does not change, such as hyper-parameters or settings.

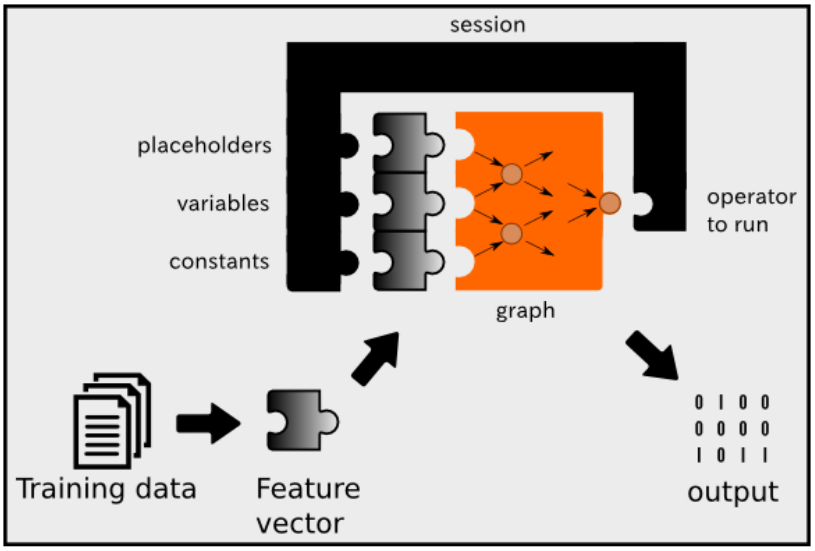


Figure 2.4 The session dictates how the hardware will be used to most efficiently process the graph. When the session starts, it assigns the CPU and GPU devices to each of the nodes. After processing, the session outputs data in a usable format, such as a NumPy array. A session optionally may be fed placeholders, variables, and constants.

2.5 Writing code in Jupyter

Because TensorFlow is primarily a Python library, we should make full use of Python's interpreter. Jupyter is a mature environment to exercise the interactive nature of the language. It is a web application that displays computation elegantly so that you can share annotated interactive algorithms with others to teach a technique or demonstrate code.

You can share your Jupyter notebooks with others to exchange ideas and download other's to learn about their code. See the appendix to get started with installing the Jupyter notebook.

From a new terminal, change directory to where you want to practice TensorFlow code, and start a notebook server.

```
$ cd ~/MyTensorFlowStuff
$ jupyter notebook
```

Running the previous command should launch a new browser window with the Jupyter notebook dashboard. If no window automatically opens up, you can manually navigate to http://localhost:8888 from any browser.

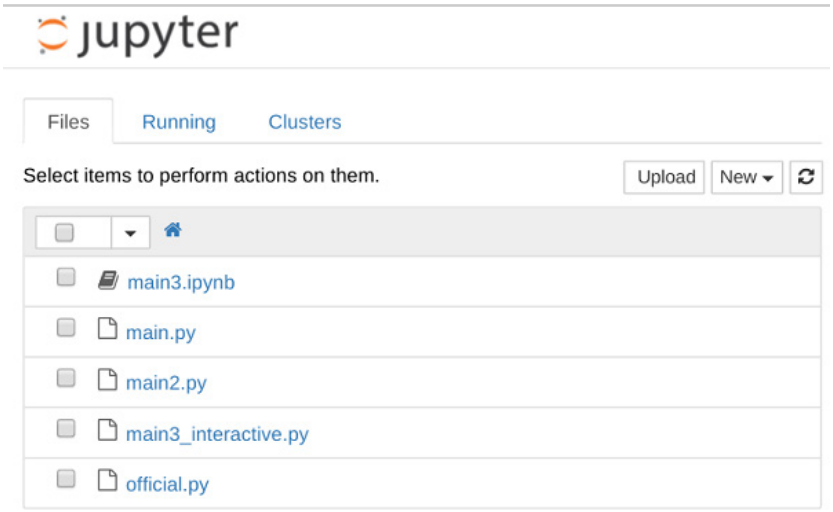


Figure 2.5 Running Jupyter notebook will launch an interactive notebook on http://localhost:8888.

Create a new notebook by clicking the dropdown menu on the top right labeled "New", under "Notebooks", choose "Python 2". This creates a new file called "Untitled.ipynb" which you can immediately start editing through the browser interface. You can change the name of the notebook by clicking the current "Untitled" name and typing in something more memorable, such as "TensorFlow Example Notebook."

Everything in the Jupyter notebook is an independent chunk of code or text called a cell. Then help divide a long block of code into manageable pieces of code snippets and

documentation. You can run cells individually, or choose to run everything at once, in order. There the three common ways to evaluate cells:

4. Pressing `Shift+Enter` on a cell executes the cell and highlights the next cell below.
5. Pressing `Ctrl+Enter` will maintain the cursor on the current cell after executing it.
6. And last, pressing `Alt+Enter` will execute the cell and then insert a new empty cell directly below.

You can change the cell type by clicking the dropdown in the toolbar as seen in figure 2.6. Alternatively, you can press `Esc` to leave edit mode, use the arrow keys to highlight a cell, and hit `Y` to change it to code mode or `M` for markdown mode.

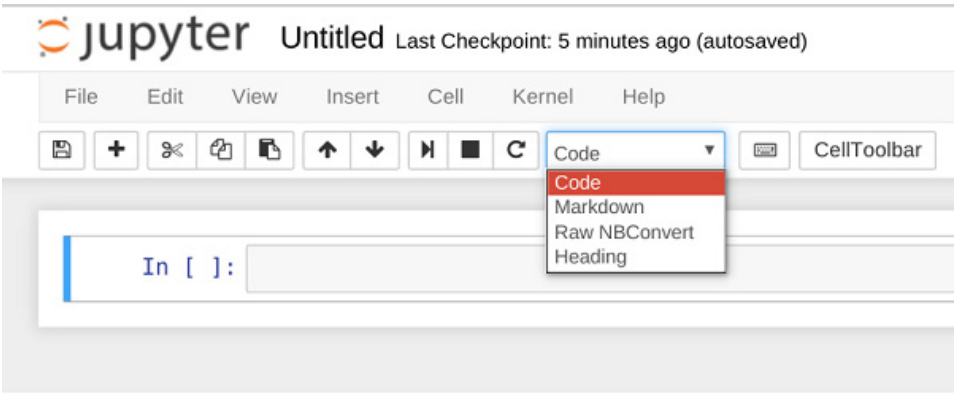


Figure 2.6 The dropdown menu changes the type of cell in the notebook. The "Code" cell is for Python code, whereas the "Markdown" code is for text descriptions.

Finally, we can create a Jupyter notebook that elegantly demonstrates some TensorFlow code by interlacing code and text cells as follows in figure 2.7.

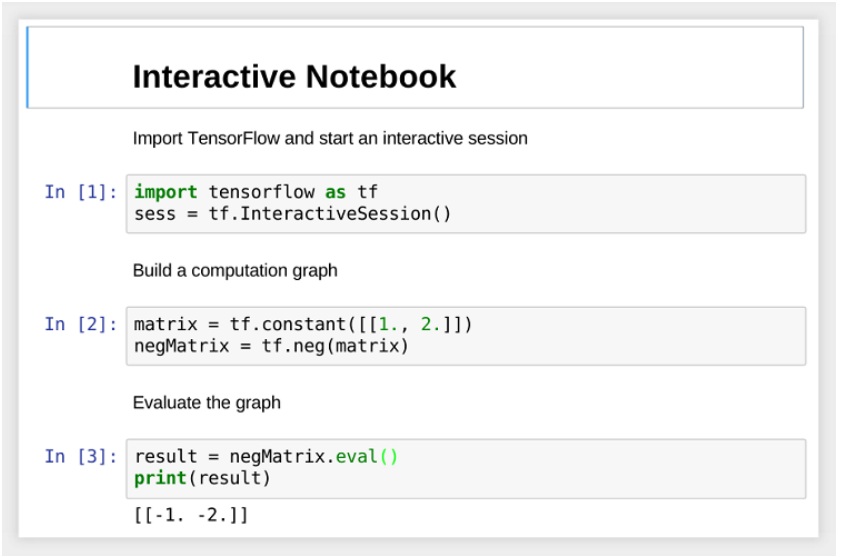


Figure 2.7 An interactive Python notebook presents both code and comments side by side.

2.6 Using variables

Using TensorFlow constants is a good start, but most interesting applications require data to change. For example, a neuroscientist may be interested in detecting neural activity from sensor measurements. A spike in neural activity could be a Boolean variable that changes over time. To capture this in TensorFlow, you can use the Variable class to represent a node whose value changes over time.

Example of using a Variable object in Machine Learning

Finding the equation of a line that best fits many points is a classic machine learning problem that will be discussed in greater detail in the next chapter. Essentially, the algorithm starts with an initial guess, which is an equation characterized by a few numbers (such as the slope or y-intercept). Over time, the algorithm keeps generating a better and better guess for these numbers, which are also called *parameters*.

So far we've only been manipulating constants. For this reason TensorFlow allows richer tools such as variables, which as you know are containers for values that may change over time. A machine learning algorithm updates the parameters of a model until it finds the optimal value. In the world of machine learning, it is common for parameters to fluctuate until eventually settling down. Therefore, variables are an excellent choice of data structure for them.

The code in listing 2.9 is a simple TensorFlow program that demonstrates how to use variables. It updates a variable whenever sequential data abruptly increases in value. Think about recording measurements of a neuron's activity over time. This piece of code can detect

when the neuron's activity suddenly spikes. Of course, the algorithm is an oversimplification for didactic purposes.

Start with importing TensorFlow. Optionally, declare the session using `tf.InteractiveSession()` so that the session variable doesn't need to be passed around. This makes writing code in Jupyter notebooks easier.

Listing 2.9 Using a variable

```
import tensorflow as tf
sess = tf.InteractiveSession()      ❶


raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]   ❷
spike = tf.Variable(False)          ❸
spike.initializer.run()             ❹


for i in range(1, len(raw_data)):   ❺
    if raw_data[i] - raw_data[i-1] > 5:
        updater = tf.assign(spike, True)  //F
        updater.eval()  //F
    else:
        tf.assign(spike, False).eval()
    print("Spike", spike.eval())

sess.close()    ❼
```

❶ Start the session in interactive mode so we won't need to pass around sess
❷ Let's say we have some raw data like this
❸ Create a Boolean variable called spike to detect a sudden increase in a series of numbers
❹ Because all variables must be initialized, initialize the variable by called run() on its initializer
❺ Loop through the data and update the spike variable when there is a significant increase
❻ To update a variable, assign it a new value using tf.assign(old, new). Evaluate it to see the change.
❼ Remember to close the session after it will no longer be used

The expected output of listing 2.9 is a list of spike values over time:

```
('Spike', False)
('Spike', True)
('Spike', False)
('Spike', False)
('Spike', True)
('Spike', False)
('Spike', True)
```

2.7 Saving and Loading Variables

Imagine writing a monolithic block of code, yet you'd like to individually test a tiny segment. In complicated machine learning situations, saving and loading data at known checkpoints makes it much easier to debug code. TensorFlow provides an elegant interface to save and load variable values to disk. I'm going to show you how to use it for that purpose.

Let's revamp the code that you created in listing 2.9 to save the spike data to disk so you can load it elsewhere. We'll change the spike variable from a simple Boolean to a vector of Booleans that captures the history of spikes (listing 2.10). Notice that we will explicitly name the variables so they can be loaded later. Try running this code to see the results.

Listing 2.10 Saving variables

```
import tensorflow as tf                                    (1)
sess = tf.InteractiveSession()                             (1)


raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]              (2)
spikes = tf.Variable([False] * len(raw_data), name='spikes')  (3)
spikes.initializer.run()                                   (4)

saver = tf.train.Saver()                                   (5)

for i in range(1, len(raw_data)):                          (6)
    if raw_data[i] - raw_data[i-1] > 5:                    (6)
        spikes_val = spikes.eval()                         (7)
        spikes_val[i] = True                               (7)
        updater = tf.assign(spikes, spikes_val)            (7)
        updater.eval()                                     (8)


save_path = saver.save(sess, "spikes.ckpt")                (9)
print("spikes data saved in file: %s" % save_path)         (10)


sess.close()
```

(1) Import TensorFlow and enable interactive sessions
(2) Let's say we have a series of data like this
(3) Define a boolean vector called spike to locate a sudden spike in raw data
(4) Don't forget to initialize the variable
(5) The saver op will enable saving and restoring variables. If no dictionary is passed into the constructor, then the saver operators of all variables in the current program.
(6) Loop through the data and update the spike variable when there is a significant increase
(7) Update the value of spikes by using the tf.assign function
(8) Don't forget to actually evaluate the updater, otherwise spikes will not be updated
(9) Save the variable to disk
(10) Print out where the relative file path of the saved variables

You will notice a `spikes.ckpt` file in the same directory as your source code. It is a compactly stored binary file, so you cannot easily modify it with a text editor. To retrieve this data, you can use the `restore` function from the `saver` op, as demonstrated in listing 2.11.

Listing 2.11 Loading variables

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

```
spikes = tf.Variable([False]*8, name='spikes')  ❶
# spikes.initializer.run()  ❷
saver = tf.train.Saver()    ❸


saver.restore(sess, "spikes.ckpt") ❹
print(spikes.eval())        ❺


sess.close()
```

❶ Create a variable of the same size and name as the saved data
❷ You no longer need to initialize this variable because it will be directly loaded
❸ Create the saver op to restore saved data
❹ Restore data from the "spikes.ckpt" file
❺ Print the loaded data

2.8 Visualizing data using TensorBoard

In machine learning, the most time-consuming part is usually not programming, but instead it's waiting for code to finish running. For example, there is a famous dataset called ImageNet which contains over 14 million images prepared to be used in a machine learning context. Sometimes it can take up to days or weeks to finish training an algorithm using a large dataset. TensorFlow comes with a handy dashboard called *TensorBoard* for a quick peek into how values are changing in each node of the graph. That way, you can have an intuitive idea of how your code is performing.

In this section, you'll use TensorBoard to visualize how data changes. Suppose you're interested in calculating the average stock price of a company. Typically, computing the average is just a matter of adding up all the values and dividing by the total number seen, $Mean = (x_1 + x_2 + \ldots + x_n) / n$. When the total number of values is unknown, we can use a technique called *exponential averaging* to estimate the average value of an unknown number of data. The exponential average algorithm calculates the current estimated average as a function of the previous estimated average and the current value.

More succinctly, $Avg_t = f(Avg_{t-1}, x_t) = (1 - \alpha) Avg_{t-1} + \alpha x_t$. Alpha ($\alpha$) is a parameter that will be tuned, representing how strongly recent values should be biased in the calculation of the average. The higher the value of α, the most dramatically the calculated average will differ from the previously estimated average. Figure 2.8 shows how TensorBoard visualizes the values and corresponding running average over time.

When you code this, it's a good idea to think about the main piece of computation that takes place in each iteration. In our case, each iteration will compute $Avg_t = (1- \alpha) Avg_{t-1} + \alpha x_t$. As a result, we can design a TensorFlow operator (Listing 2.12) that does exactly as the formula says. To actually run this code, we'll have to eventually define `alpha`, `curr_value`, and `prev_avg`.

Listing 2.12 Defining the average update operator

```
update_avg = alpha * curr_value + (1 - alpha) * prev_avg  (1)
```

(1) alpha is a tf.constant, curr_value is a placeholder, and prev_avg is a variable

We'll define the undefined variables later. Skipping ahead, let's jump right to the session part to see how our algorithm should behave. Listing 2.13 sets up the primary loop and calls the `update_avg` operator on each iteration. Running the `update_avg` operator depends on the `curr_value`, which is fed using the `feed_dict` argument.

Listing 2.13 Running iterations of the exponential average algorithm

```
raw_data = np.random.normal(10, 1, 100)

with tf.Session() as sess:
    for i in range(len(raw_data)):
        curr_avg = sess.run(update_avg, feed_dict={curr_value:raw_data[i]}
        sess.run(tf.assign(prev_avg, curr_avg))
```

Great, the general picture is clear because all that's left to do is to write out the undefined variables. Let's fill in the gaps and implement a working piece of TensorFlow code. Copy listing 2.14 so you can run it.

Listing 2.14 Filling in missing code to complete the exponential average algorithm

```
import tensorflow as tf
import numpy as np

raw_data = np.random.normal(10, 1, 100)  (1)

alpha = tf.constant(0.05)  (2)
curr_value = tf.placeholder(tf.float32)  (3)
prev_avg = tf.Variable(0.)  (4)
update_avg = alpha * curr_value + (1 - alpha) * prev_avg

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)
    for i in range(len(raw_data)):
        curr_avg = sess.run(update_avg, feed_dict={curr_value: raw_data[i]})
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
```

(1) Create a vector of 1000 numbers with a mean of 10 and standard deviation of 1
(2) Define alpha as a constant
(3) A placeholder is just like a variable, but the value is injected from the session
(4) Initialize the previous average to some

Now that we have a working implementation of a moving average algorithm, let's visualize the results using TensorBoard. To communicate with TensorBoard, we must use a Summary op, which produces serialized string used by a `SummaryWriter` to save updates to a directory.

Every time you call the `add_summary` method from the `SummaryWriter`, TensorFlow will save data to disk for TensorBoard to use.

Run the following command to make a directory called "logs" in the same folder as this source code.

```
$ mkdir logs
```

Run TensorBoard with the location of the "logs" directory passed in as an argument:

```
$ tensorboard --logdir=./logs
```

Open a browser and navigate to http://localhost:6006, which is the default URL for TensorBoard. Listing 2.15 shows you how to hook up the `SummaryWriter` to your code. Run it and refresh the TensorBoard to see the visualizations.

Listing 2.15 Writing summaries to view in TensorBoard

```
import tensorflow as tf
import numpy as np

raw_data = np.random.normal(10, 1, 100)

alpha = tf.constant(0.05)
curr_value = tf.placeholder(tf.float32)
prev_avg = tf.Variable(0.)
update_avg = alpha * curr_value + (1 - alpha) * prev_avg

avg_hist = tf.scalar_summary("running average", update_avg)     ❶
value_hist = tf.scalar_summary("incoming values", curr_value)  ❷
merged = tf.merge_all_summaries()             ❸
writer = tf.train.SummaryWriter("./logs")     ❹
init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)
    for i in range(len(raw_data)):
        summary_str, curr_avg = sess.run([merged, update_avg], feed_dict={curr_value:
       raw_data[i]})  ❺
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
        writer.add_summary(summary_str, i)    ❻
```

❶ Create a summary node for the averages
❷ Create a summary node for the values
❸ Merge the summaries to make it easier to run together
❹ Pass in the "logs" directory location to the writer
❺ Run the merged op and the update_avg op at the same time
❻ Add the summary to the writer

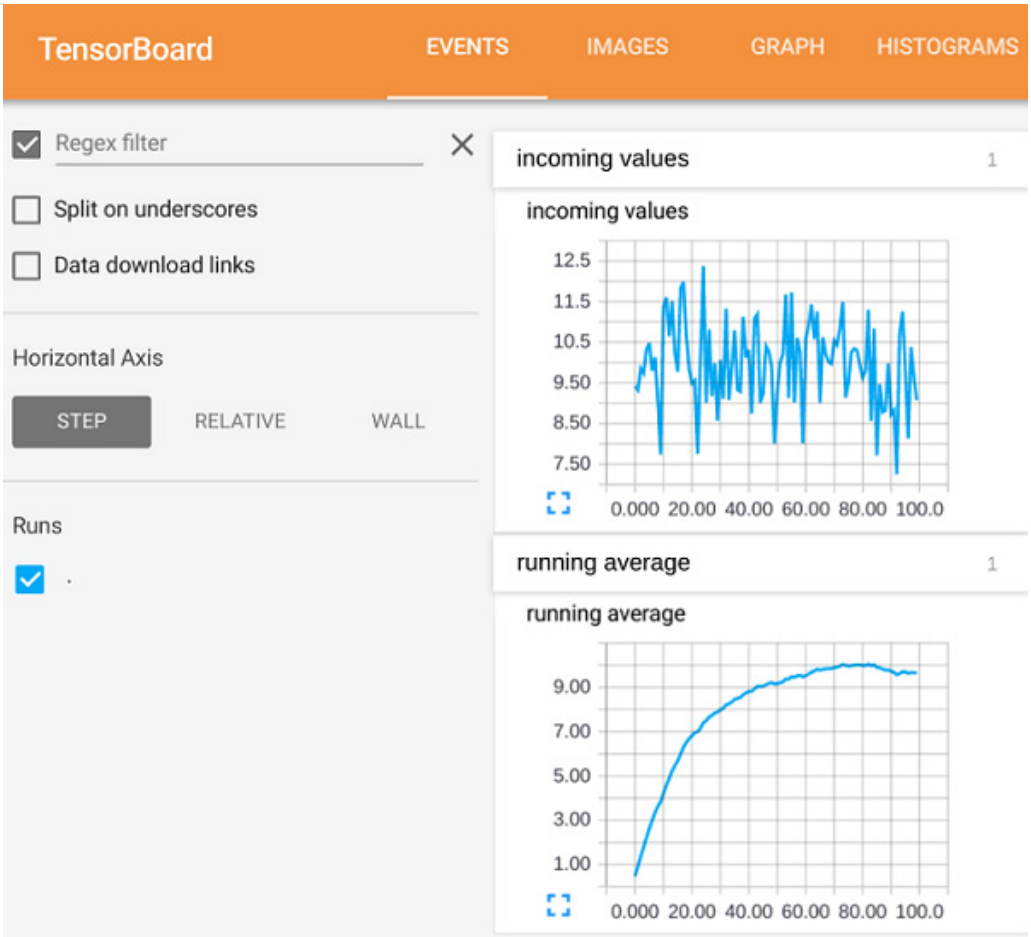


Figure 2.8 TensorBoard provides a user-friendly interface to visualize data produced in TensorFlow.

2.9 Summary

You're now ready to use TensorFlow for real-world machine learning. Here are some key concepts to remember from this chapter.

- You should start thinking of mathematical algorithms in terms of a flowchart of computation. When you consider each node as an operation, and edges as data flow, writing TensorFlow code becomes trivial. Once you define your graph, you evaluate it under a session, and you have your result.
- No doubt, there's more to TensorFlow than simply representing computations as a graph. As you'll come to see in the coming chapters, some of the built-in functions are

very inviting to the field of machine learning. In fact, it has some of the best support for convolutional neural networks, a recently popular type of model for processing images (with promising results in audio and text as well).

- TensorBoard provides an easy way to visualize how data changes in TensorFlow code as well as troubleshoot bugs by inspecting trends in data.
- TensorFlow works wonderfully with Jupyter notebooks, which are an elegant interactive medium to share and document Python code.

3

Linear regression and beyond

This chapter covers

- Fitting a line to data points
- Fitting arbitrary curves to data points
- Testing performance of these regression algorithms
- Applying regression to real-world data

Remember science courses back in grade school? It might have been a while ago, or who knows - maybe you're in grade school now starting your journey in machine learning early. Either way, whether you took biology, chemistry, or physics, a common technique to analyze data is to plot how changing one variable affects the other.

Imagine plotting the correlation between rainfall frequency and agriculture production. You may observe that an increase in rainfall produces an increase in agriculture rate. Fitting a line to these data points enables you to make predictions about the agriculture rate under different rain conditions. If you discover the underlying function from a few data points, then that learned function empowers you to make predictions about the values of unseen data.

Regression is a study of how to best fit a curve to summarize your data. It is one of the most powerful and well-studied types of supervised learning algorithms. In regression, we try to understand the data points by discovering the curve that might have generated them. In doing so, we seek an explanation for why the given data is scattered the way it is. The best fit curve gives us a model for explaining how the dataset might have been produced.

This chapter will show you how to formulate a real world problem to use regression. As you'll see, TensorFlow is just the right tool that endows us with some of the most powerful predictors.

3.1 Formal notation

If you have a hammer, every problem looks like a nail. This chapter will demonstrate the first major machine learning tool, regression, and formally define it using precise mathematical symbols. Learning regression first is a great idea, because many of the skills you will develop carry over to other types of problems discussed in future chapters. By the end of this chapter, regression will become the "hammer" in your box of machine learning tools.

Let's say we have data about how much money people spent on bottles of beer. Alice spent $2 on 1 bottle, Bob spent $4 on 2 bottles, and Clair spent $6 on 3 bottles. We want to find an equation that describes how the number of bottles changes the total cost. For example, if every beer bottle costs $2, then a linear equation $y = 2x$ can describe the cost of buying a particular number of bottles.

When a line appears to fit some data points well, we can claim our linear model performs well. Actually, we could have tried out many possible slopes instead of choosing the value *2*. The choice of slope is the *parameter*, and the equation produced is the *model*. Speaking in

machine learning terms, the equation of the best fit curve comes from learning the parameters of a model.

As another example, the equation $y = 3x$ is also a line, except with a steeper slope. You can replace that coefficient with any real number, let's call it w, and the equation will still produce a line: $y = wx$. Figure 3.1 shows how changing the parameter *w* affects the model. The set of all equations we can generate this way is denoted $M = \{y = wx \mid w \in \mathbb{R}\}$.

It is read "All equations $y = wx$ such that w is a real number."

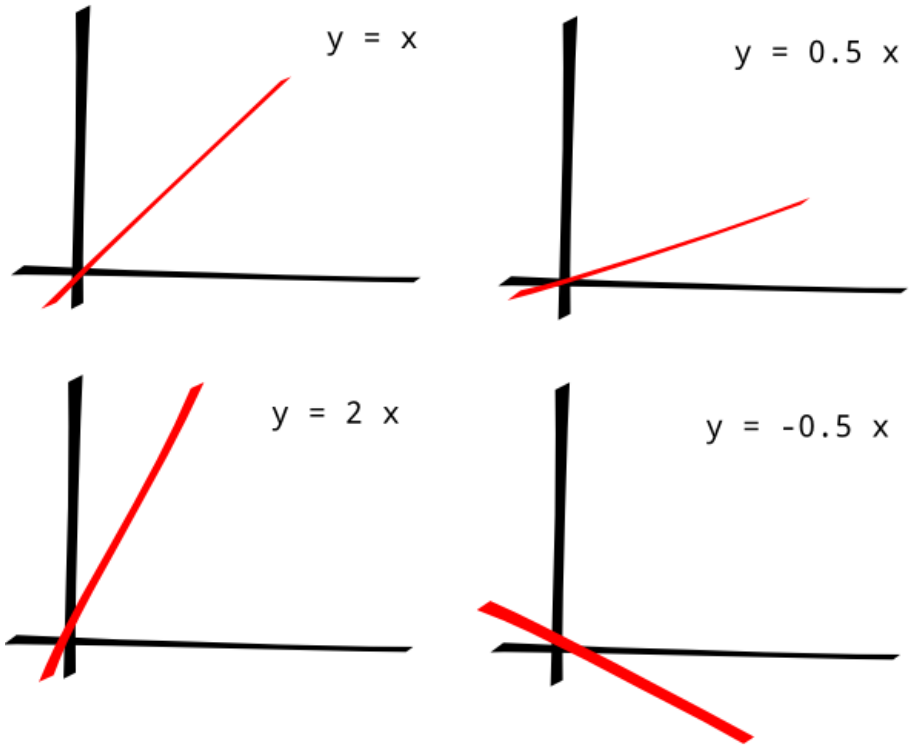


Figure 3.1 Different values of the parameter w result in different linear equations. The set of all these linear equations is what constitutes the linear model M.

M is a set of all possible *models*. Choosing a value for w generates a candidate model $M(w): y = wx$. The regression algorithms that we will write in TensorFlow will iteratively converge to better and better values for the model's parameter w. An optimal parameter, let's call it w^* (pronounced *w star*), is the best-fit equation $M(w^*): y = w^*x$.

In the most general sense, a regression algorithm tries to design a function, let's call it f, that maps an input to an output. The function's domain is a real-valued vector $\mathbb{R}^d$ and its range is the set of real numbers $\mathbb{R}$. The input of the function could be continuous or discrete. However, the output must be continuous, as demonstrated by figure 3.2.

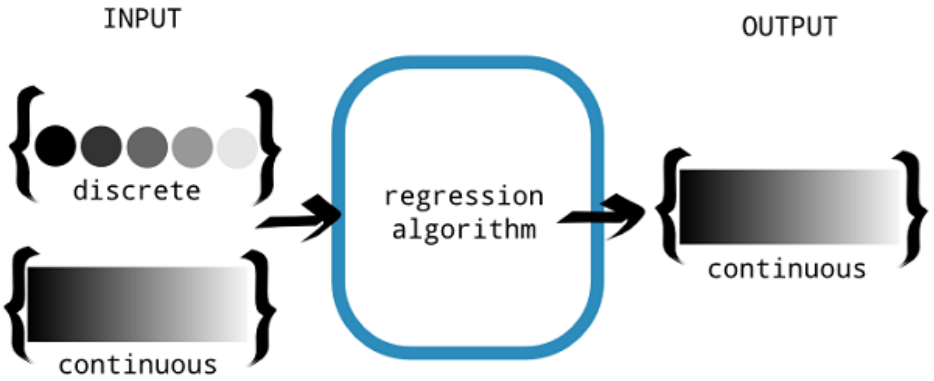


Figure 3.2 A regression algorithm is meant to produce continuous output. The input is allowed to be discrete or continuous. This distinction is important because discrete-valued outputs are handled better by classification, which is discussed in the next chapter.

BY THE WAY Regression predicts continuous outputs, but sometimes that's overkill. Sometimes we just want to predict a discrete output, such as 0 or 1, but nothing in-between. Classification is a technique better suited for such tasks, and will be discussed in Chapter 4.

We would like to discover a function `f` that agrees well with the given data points, which are essentially input/output pairs. Unfortunately, the number of possible functions is infinite, so we'll have no luck trying them out one-by-one. Having too many options available to choose from is usually a bad idea. It behooves us to tighten the scope of all the functions we want to deal with. For example, if we look at only straight lines to fit a set of data points, then the search becomes much easier.

EXERCISE 3.1 How many possible functions exist that map 10 integers to 10 integers? For example, let `f(x)` be a function that can take numbers 0 through 9 and produce numbers 0 through 9. One example is the identity function that mimics its input, for instance `f(0) = 0, f(1) = 1`, and so on. How many other functions exist?

ANSWER 10^10 = 10,000,000,000

3.1.1 How do you know the regression algorithm is working?

Let's say we're trying to sell a housing market predictor algorithm to a real estate firm. It predicts housing prices given some properties such as number of bedrooms and lot-size. Real estate companies can easily make millions with such information, but they need some proof that it actually works before buying the algorithm from you.

To measure the success of the learning algorithm, you'll need to understand two important concepts: *variance* and *bias*.

- Variance is how sensitive a prediction is to what training set was used. Ideally, how we choose the training set shouldn't matter – meaning a lower variance is desired.
- Bias is the strength of assumptions made about the training dataset. Making too many assumptions might make it hard to generalize, so we prefer low bias as well.

If a model is too flexible, it may accidentally memorize the training data instead of resolving useful patterns. You can imagine a curvy function passing through every point of a dataset, appearing to produce no error. If that happens, we say the learning algorithm *overfits* the data. In this case, the best-fit curve will agree with the training data well; however, it may perform abysmally when evaluated on the testing data (see figure 3.3).

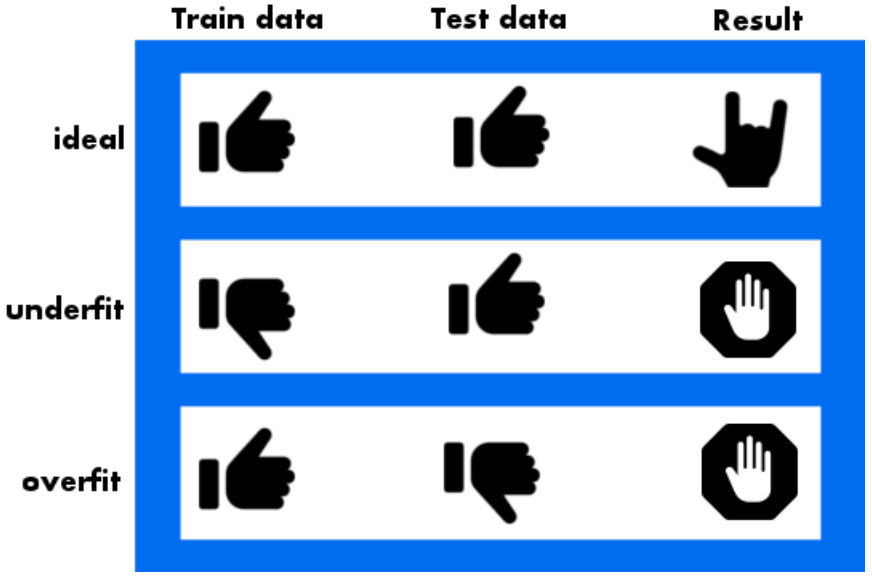


Figure 3.3 Ideally, the best fit curve fits well on both the training data as well as the test data. However, if we witness it fitting the test data much better than the training data, there's a chance that our model is underfitting. Lastly, if it performs poorly on the test data but well on the training data, then we know the model is overfitting.

On the other hand, a not-so-flexible model may generalize better to unseen testing data, but would score relatively low in the training data. That situation is called *underfitting*. A too flexible model has high variance and low bias, whereas a too strict model has low variance and high bias. Ideally we would like a model with both low variance error and low bias error. That way, it both generalizes to unseen data and captures the regularities of the data. See figure 3.4 for examples.

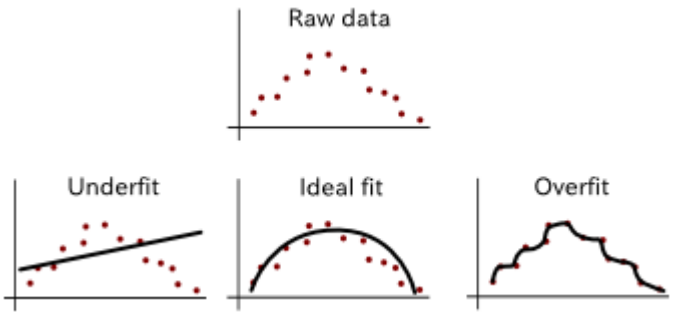


Figure 3.4 Examples of under-fitting and over-fitting the data.

Concretely, the variance of a model is a measure of how badly the responses fluctuate, and the bias is a measure of how badly the response is offset from the ground-truth. You want your model to achieve both accurate (low bias) as well as reproducible (low variance) results.

EXERCISE 3.2 Let's say our model is `M(w): y = wx`. How many possible functions can you generate if the values of weight parameters `w` must be integers between *0* and *9* (inclusive)?

ANSWER Only *10*. Namely, *{y=0, y=x, y=2x, ..., y=9x}*.

To measure success in machine learning, we partition the dataset into two groups: a training dataset, and a testing dataset. The model is learned using the training dataset, and performance is evaluated on the testing dataset (exactly how we evaluate performance will be described in the next section). Out of the many possible weight parameters we can generate, the goal is to find one that best fits the data. The way we measure "best fit" is by defining a cost function, which is discussed in greater detail in section 3.2

3.2 Linear Regression

Let's start by creating fake data to leap into the heart of linear regression. Create a Python source file called `regression.py` and follow along with listing 3.1 to initialize data. The code will produce an output similar to figure 3.5.

Listing 3.1 Visualizing raw input

```
import numpy as np  ❶
import matplotlib.pyplot as plt    ❷

x_train = np.linspace(-1, 1, 101)  ❸
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33  ❹

plt.scatter(x_train, y_train)      ❺
plt.show()          ❺
```

1. Import NumPy to help generate initial raw data
2. Use matplotlib to visualize the data
3. The input values are 101 evenly spaced numbers between -1 and 1
4. The output values are proportional to the input but with added noise
5. Use matplotlib's function to generate a scatter plot of the data

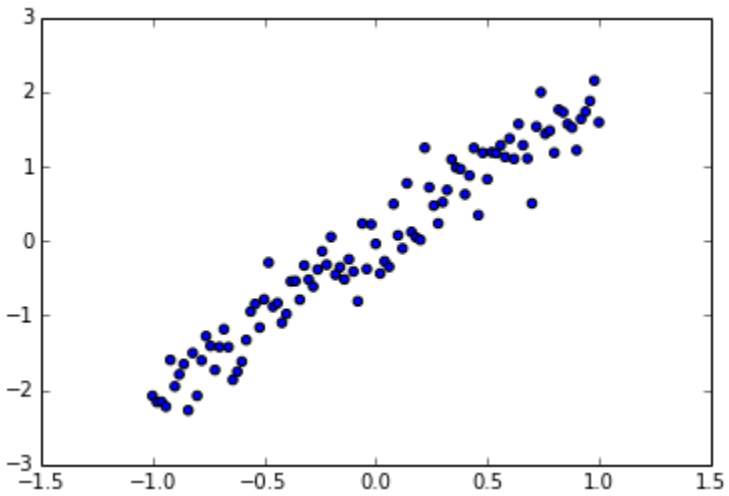


Figure 3.5 Scatter plot of y = x + noise.

Now that you have some data points available, you can try fitting a line. At the very least, you need to provide TensorFlow with a score for each candidate parameter it tries. This score assignment is commonly called a *cost function*. The higher the cost, the worse the model parameter will be. For example, if the best fit line is `y = 2x`, a parameter choice of *2.01* should have low cost, but the choice of *-1* should have higher cost.

After we define the situation as a cost minimization problem, as denoted in figure 3.6, TensorFlow takes care of the inner workings and tries to update the parameters in an efficient way to eventually reach the best possible value. Each step of updating the parameters is called an *epoch*.

$$w^* = \arg min_w \underbrace{cost(Y_{model}, Y_{ideal})}_{\underbrace{|Y_{model} - Y_{ideal}|}_{M(w,X)}}$$

Figure 3.6 Whichever parameter `w` minimizes, the cost is optimal. Cost is defined as the norm of the error between the ideal value with the model response. And lastly, the response value is calculated from the function in the model set.

In this example, the way we define cost is by the sum of errors. The error in predicting x is often calculated by the squared difference between the actual value `f(x)` and the predicted value `M(w, x)`. Therefore, cost is the sum of squared differences between the actual and predicted values, as seen by figure 3.7.

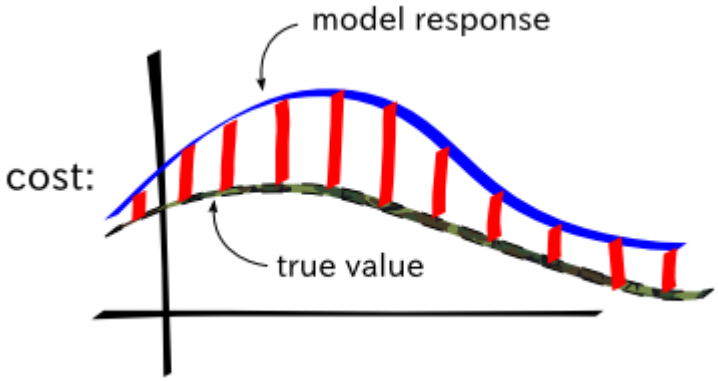


Figure 3.7 The cost is the norm of the point-wise difference between the model response and the true value.

Let's update our previous code to look like listing 3.2. This code defines the cost function, and asks TensorFlow to run an optimizer to find the optimal solution for the model parameters.

Listing 3.2 Solving linear regression

```
import tensorflow as tf  (1)
import numpy as np       (1)
import matplotlib.pyplot as plt         (1)

learning_rate = 0.01     (2)
training_epochs = 100    (2)

x_train = np.linspace(-1, 1, 101)       (3)
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33             (3)

X = tf.placeholder("float")             (4)
Y = tf.placeholder("float")             (4)

def model(X, w):         (5)
    return tf.mul(X, w)

w = tf.Variable(0.0, name="weights")    (6)

y_model = model(X, w)    (7)
cost = (tf.square(Y-y_model))           (7)

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)  (8)

sess = tf.Session()      (9)
init = tf.initialize_all_variables()    (9)
sess.run(init)           (9)

for epoch in range(training_epochs):    (10)
  for (x, y) in zip(x_train, y_train):  (11)
    sess.run(train_op, feed_dict={X: x, Y: y})  (12)
  w_val = sess.run(w)    (13)

sess.close()             (14)
plt.scatter(x_train, y_train)           (15)
y_learned = x_train*w_val               (16)
plt.plot(x_train, y_learned, 'r')       (16)
plt.show()               (16)
```

(1) Import TensorFlow for the learning algorithm. We'll need NumPy to set up the initial data. And we'll use matplotlib to visualize our data.
(2) Define some constants used by the learning algorithm. There are called hyper-parameters.
(3) Set up fake data that we will use to find a best fit line
(4) Set up the input and output nodes as placeholders since the value will be injected by x_train and y_train.
(5) Define the model as y = w*x
(6) Set up the weights variable
(7) Define the cost function
(8) Define the operation that will be called on each iteration of the learning algorithm
(9) Set up a session and initialize all variables
(10) Loop through the dataset multiple times
(11) Loop through each item in the dataset
(12) Update the model parameter(s) to try to minimize the cost function
(13) Obtain the final parameter value
(14) Close the session
(15) Plot the original data

16 **Plot the best fit line**

Congratulations, you've just solved linear regression using TensorFlow! Conveniently, the rest of the topics in regression are just minor modifications of Listing 3.2. The entire pipeline involves updating model parameters using TensorFlow as summarized in figure 3.8.

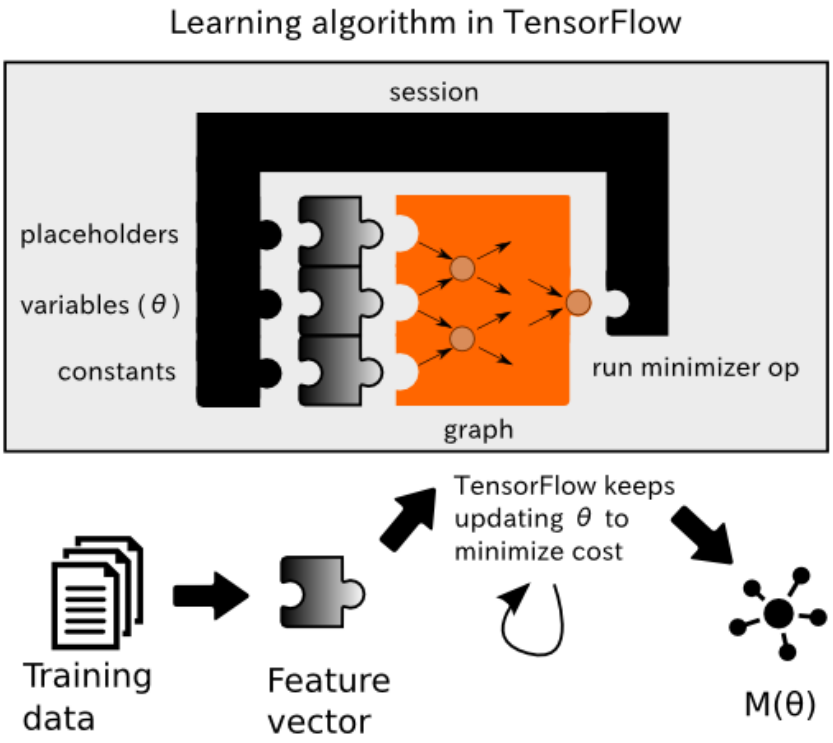


Figure 3.8 The learning algorithm updates the model's parameters to minimize the given cost function.

3.3 Polynomial Model

Linear models may be an intuitive first guess, but rarely are real-world correlations so simple. For example, the trajectory of a missile through space is curved relative to the observer on Earth. WiFi signal strength degrades with an inverse square law. The height of a flower over its lifetime is certainly not linear.

When data points appear to form smooth curves rather than straight lines, we need to change our regression model from a straight line to something else. One such approach is to use a polynomial model. A polynomial is a generalization of a linear function. The n^{th} degree polynomial looks like the following:

$$f(x) = w_n x^n + \ldots + w_2 x^2 + w_1 x + w_0$$

ASIDE When n = 1, a polynomial is simply a linear equation $f(x) = w_1 x + w_0$.

Consider the scatter plot in figure 3.8, showing the input on the x-axis and the output on the y-axis. As you can tell, a straight line is insufficient to describe all the data. A polynomial function is a more flexible generalization of a linear function.

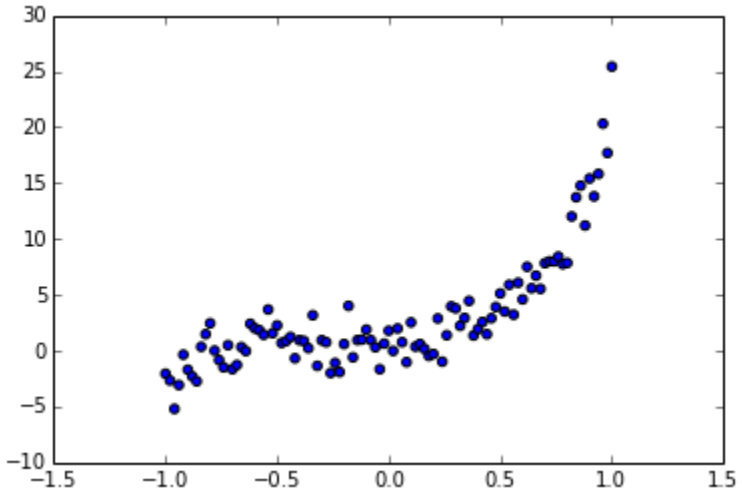


Figure 3.9 Data points like this are not suitable for a linear model.

Let's try to fit a polynomial to this kind of data. Create a new file called `polynomial.py` and follow along to listing 3.3.

Listing 3.3 Using a polynomial model

```
import tensorflow as tf  (1)
import numpy as np       (1)
import matplotlib.pyplot as plt  (1)

learning_rate = 0.01     (1)
training_epochs = 40     (1)

trX = np.linspace(-1, 1, 101)    (2)

num_coeffs = 6           (3)
trY_coeffs = [1, 2, 3, 4, 5, 6]  (3)
trY = 0                  (3)
for i in range(num_coeffs):      (3)
    trY += trY_coeffs[i] * np.power(trX, i) (3)

trY += np.random.randn(*trX.shape) * 1.5    (4)

plt.scatter(trX, trY)    (5)
plt.show()               (5)
```

```
X = tf.placeholder("float")       ❻
Y = tf.placeholder("float")       ❻

def model(X, w):                  ❼
    terms = []                    ❼
    for i in range(num_coeffs):   ❼
        term = tf.mul(w[i], tf.pow(X, i))   ❼
        terms.append(term)        ❼
    return tf.add_n(terms)        ❼

w = tf.Variable([0.] * num_coeffs, name="parameters")  ❽
y_model = model(X, w)             ❽

cost = (tf.pow(Y-y_model, 2))     ❾
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) ❾

sess = tf.Session()               ❿
init = tf.initialize_all_variables()        ❿
sess.run(init)                    ❿

for epoch in range(training_epochs):        ❿
    for (x, y) in zip(trX, trY):            ❿
        sess.run(train_op, feed_dict={X: x, Y: y})     ❿

w_val = sess.run(w)               ❿
print(w_val)                      ❿

sess.close()                      ⓫

plt.scatter(trX, trY)             ⓬
trY2 = 0                          ⓬
for i in range(num_coeffs):       ⓬
    trY2 += w_val[i] * np.power(trX, i)     ⓬
plt.plot(trX, trY2, 'r')          ⓬
plt.show()                        ⓬
```

❶ Import the relevant libraries and initialize the hyper-parameters
❷ Set up some fake raw input data
❸ Set up raw output data based on a degree 5 polynomial
❹ Add some noise
❺ Show a scatter plot of the raw data
❻ Define the nodes to hold values for input/output pairs
❼ Define our polynomial model
❽ Set up the parameter vector to all zeros
❾ Define the cost function just as before
❿ Set up the session and run the learning algorithm just as before
⓫ Close the session when done
⓬ Plot the result

The final output of listing 3.3 is a 5^{th}-degree polynomial that fits the data, as seen in figure 3.9.

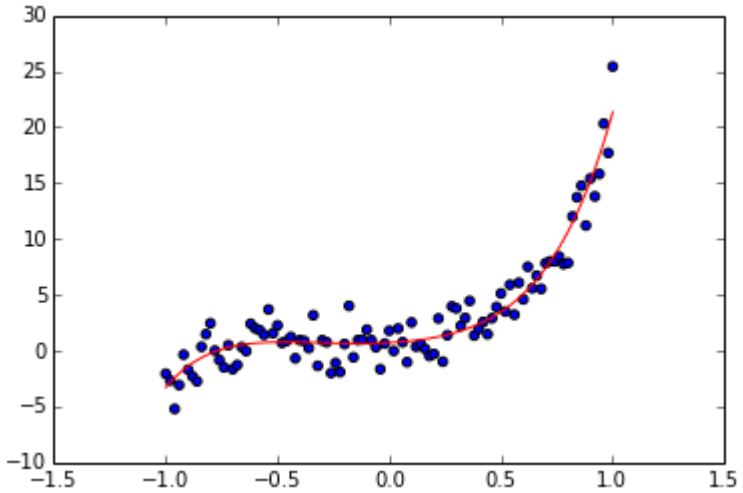


Figure 3.10 The best fit curve smoothly aligns with the nonlinear data

3.4 Regularization

Don't be fooled by the wonderful flexibility of polynomials, as seen in the previous section. Just because higher order polynomials are extensions of lower ones doesn't mean we should always prefer to use the more flexible model.

In the real world, raw data rarely forms a smooth curve mimicking a polynomial. Imagine we're plotting house prices over time. The data likely will contain fluctuations. The goal of regression is to represent the complexity in a simple mathematical equation. If our model is too flexible, the model may be over-complicating its interpretation of the input.

Take for example the data presented in figure 3.10. We try to fit an 8^{th} order polynomial into points that appear to simply follow the equation $y = x^2$. This process fails miserably as the algorithm tries its best to update the 9 coefficients of the polynomial.

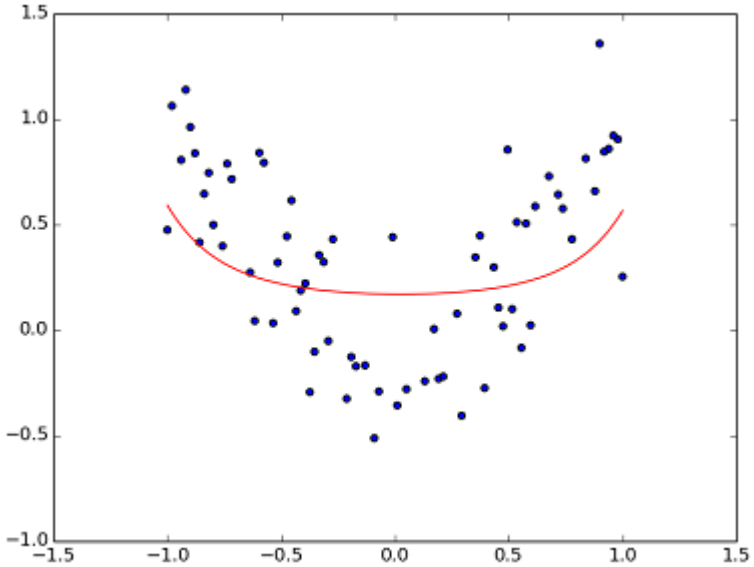


Figure 3.10 When the model is too flexible, a best fit curve could look awkwardly complicated. We need to use regularization to improve the fit.

Regularization is a technique to structure the parameters in a form we prefer, often to solve the problem of overfitting. In our case, we anticipate the learned coefficients to be *0* everywhere except for the 2^{nd} term, thus producing the curve $y = x^2$. The regression algorithm has no idea about this, so it may produce curves that score well but look strangely over-complicated.

To influence the learning algorithm to produce a smaller coefficient vector (let's call it w), we add that penalty to the loss term. To control the how significantly we want to weigh the penalty term, we actually multiply the penalty by a constant non-negative number, λ, as follows:

$$Cost(X, Y) = Loss(X, Y) + \lambda |w|$$

If λ is set to *0*, then regularization is not in play. As we set λ to larger and larger values, parameters with larger norms will be heavily penalized. The choice of norm depends case by case, but typically, the parameters are measured by their L1 or L2 norm. Simply put, regularization reduces some of the flexibility of the otherwise easily tangled model.

To figure out which value of the regularization parameter λ performs best, we must split our dataset into two disjointed sets. About 70% of the randomly chosen input/output pairs will consist of the training dataset. The remaining 30% will be used for testing. We will use the function provided in listing 3.4 for splitting the dataset.

Listing 3.4 Splitting the dataset into testing and training

```
def split_dataset(x_dataset, y_dataset, ratio):          (1)
    arr = np.arange(x_dataset.size)          (2)
    np.random.shuffle(arr)  (2)
    num_train = ratio * x_dataset.size       (3)
    x_train = x_dataset[arr[0:num_train]]    (4)
    y_train = y_dataset[arr[0:num_train]]    (4)
    x_test = x_dataset[arr[num_train:x_dataset.size]] (5)
    y_test = y_dataset[arr[num_train:x_dataset.size]] (5)
return x_train, x_test, y_train, y_test      (6)
```

(1) Take the input and output dataset as well as the desired split ratio
(2) Shuffle a list of numbers
(3) Calculate the number of training examples
(4) Use the shuffled list to split the x_dataset
(5) Likewise, split the y_dataset
(6) Return the split x and y datasets

With this handy tool, we can begin testing which value of λ performs best on our data. Open up a new python file and follow along with listing 3.5.

Listing 3.5 Evaluating regularization parameters

```
import tensorflow as tf                          (1)
import numpy as np                               (1)
import matplotlib.pyplot as plt                  (1)

learning_rate = 0.001        (1)
training_epochs = 1000       (1)
reg_lambda = 0.              (1)

x_dataset = np.linspace(-1, 1, 100)          (2)

num_coeffs = 9               (2)
y_dataset_params = [0.] * num_coeffs         (2)
y_dataset_params[2] = 1      (2)
y_dataset = 0                (2)
for i in range(num_coeffs): (2)
    y_dataset += y_dataset_params[i] * np.power(x_dataset, i)  (2)
y_dataset += np.random.randn(*x_dataset.shape) * 0.3           (2)

(x_train, x_test, y_train, y_test) = split_dataset(x_dataset, y_dataset, 0.7)  (3)

X = tf.placeholder("float") (4)
Y = tf.placeholder("float") (4)

def model(X, w):             (5)
    terms = []               (5)
    for i in range(num_coeffs):              (5)
        term = tf.mul(w[i], tf.pow(X, i))    (5)
        terms.append(term)   (5)
    return tf.add_n(terms)   (5)

w = tf.Variable([0.] * num_coeffs, name="parameters")  (6)
y_model = model(X, w)        (6)
```

```
cost = tf.div(tf.add(tf.reduce_sum(tf.square(Y-y_model)),
                     tf.mul(reg_lambda, tf.reduce_sum(tf.square(w)))),
              2*x_train.size)              ❻
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) ❻

sess = tf.Session()  ❼
init = tf.initialize_all_variables()       ❼
sess.run(init)       ❼

for reg_lambda in np.linspace(0,1,100):    ❽
    for epoch in range(training_epochs):   ❽
        sess.run(train_op, feed_dict={X: x_train, Y: y_train})    ❽
    final_cost = sess.run(cost, feed_dict={X: x_test, Y:y_test})  ❽
    print('reg lambda', reg_lambda)        ❽
    print('final cost', final_cost)        ❽

sess.close()         ❾
```

❶ Import the relevant libraries and initialize the hyper-parameters
❷ Create a fake dataset. $y = x^2$
❸ Split the dataset into 70% training and testing 30%
❹ Set up the input/output placeholders
❺ Define our model
❻ Define the regularized cost function
❼ Set up the session
❽ Try out various regularization parameters
❾ Close the session

If we plot the corresponding output per each regularization parameter from listing 3.5, we can see how the curve changes as λ increases. When λ is 0, the algorithm favors using the higher order terms to fit the data. As we start penalizing parameters with a high L2 norm, the cost decreases, indicating that we are recovering from overfitting.

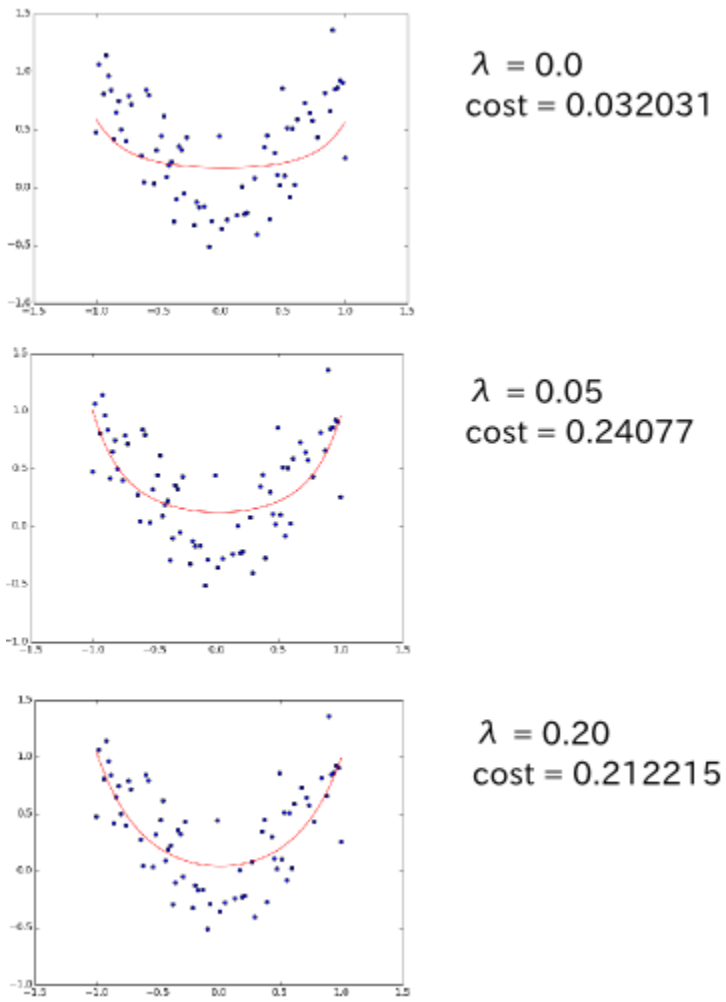


Figure 3.12 As we increase the regularization parameter to some extent, the cost decreases. This implies that the model was originally overfitting the data, and regularization helped add some structure.

3.5 Available datasets

Running linear regression on fake data is like buying a new car and never driving it. This awesome machinery begs to manifest the real world! Fortunately, many datasets are available online to test your new-found knowledge of regression.

- University of Massachusetts Amherst supplies small datasets of various types.
- http://www.umass.edu/statdata/statdata/
- Kaggle contains all types of large scale data for machine learning competitions.

- https://www.kaggle.com/datasets
- Data.gov is an open data initiative by the US government, which contains many interesting and practical datasets.
- https://catalog.data.gov

A good number of datasets contain dates. For example, there's a dataset about all phone-calls to the 3-1-1 non-emergency line in Los Angeles, California. You can obtain it here: https://data.lacity.org/dataset/311-Call-Center-Tracking-Data/ukiu-8trj. A good feature to track could be the frequency of calls per day, or week, or month. For convenience, listing 3.6 allows you to obtain a weekly frequency count of data items.

Listing 3.6 Parsing raw CSV datasets

```
import csv    (1)
import time   (2)

def read(filename, date_idx, date_parse, year, bucket=7):

    days_in_year = 365

    (3)
    freq = {}
    for period in range(0, int(days_in_year/bucket)):
        freq[period] = 0

    (4)
    with open(filename, 'rb') as csvfile:
        csvreader = csv.reader(csvfile)
        csvreader.next()
        for row in csvreader:
            t = time.strptime(row[date_idx], date_parse)
            if t.tm_year == year and t.tm_yday < (days_in_year-1):
                freq[int(t.tm_yday / bucket)] += 1

    return freq

freq = read('311.csv', 0, '%m/%d/%Y', 2014)  (5)
```

(1) For easily reading csv files
(2) For using useful date functions
(3) Set up initial frequency map
(4) Read data and aggregate count per period
(5) Obtain a weekly frequency count of 3-1-1 phone calls in 2014

3.6 Summary

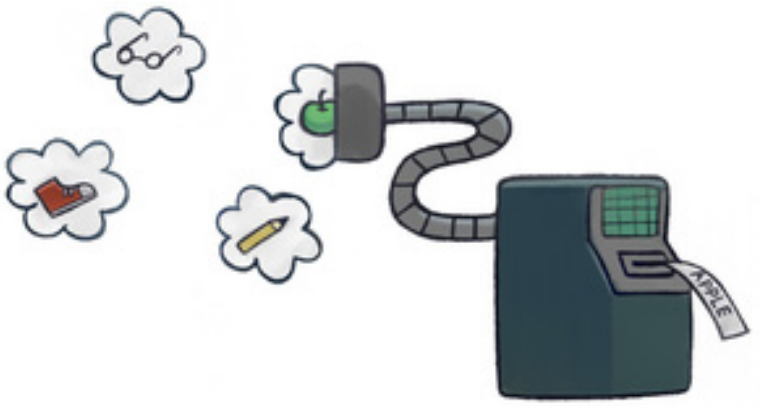
Now your toolbox of useful machine learning algorithms is no longer empty. With regression as your "hammer," you're ready to start solving real-world problems of your own. Keep in mind the following key facts covered in this chapter.

- Regression is a type of supervised machine learning for predicting continuous-valued output.

- By defining a set of models, we greatly reduce the search space of possible functions. Moreover, TensorFlow takes advantage of the differentiable property of the functions by running its efficient gradient descent optimizers to learn the parameters.
- We can easily modify linear regression to learn polynomials or other more complicated curves.
- To avoid overfitting our data, we regularize the cost function by penalizing larger valued parameters.
- If the output of the function is not continuous, a classification algorithm should instead be used (see the next Chapter).
- TensorFlow enables us to solve linear regression machine learning problems effectively and efficiently, and hence make useful predictions about important matters, such as agricultural production, heart conditions, housing prices, and more.

4

A gentle introduction to classification



This chapter covers

- Formal notation
- Logistic regression
- Type 1 and Type 2 errors
- Multiclass classification

Imagine an advertisement agency collecting information about user interactions to decide what type of ad to show. That's not so uncommon. Google, Twitter, Facebook, and other big tech giants that rely on ads have creepy-good personal profiles of their users to help deliver personalized ads. A user who's recently searched for gaming keyboards or graphics cards is probably more likely to click ads about the latest and greatest video games.

It may be difficult to cater a specially crafted advertisement for each individual, so grouping users into categories is a common technique. For example, a user may be categorized as a "gamer" to receive relevant video game related ads.

Machine learning has been the go-to tool to accomplish such as task. At the most fundamental level, machine learning practitioners want to build a tool to help them understand data. Being able to label data items into separate categories is an excellent way to characterize it for specific needs.

The previous chapter dealt with regression, which was about fitting a curve to data. If you recall, the best-fit curve is a function that takes as input a data item and assigns it a number. Creating a machine learning model that instead assigns discrete labels to its inputs is called *classification*. It is a supervised learning algorithm for dealing with discrete output. (Each discrete value is called a *class*.) The input is typically a feature vector, and the output is a class. If there are only two class labels (for example, True/False, On/Off, Yes/No), then we call this learning algorithm a binary classifier. Otherwise, it's called a *multiclass classifier*.

There are many types of classifiers, but this chapter will focus on the ones outlined in table 4.1. Each has its advantages and disadvantages, which we'll delve deeper once we start implementing each one in TensorFlow.

Table 4.1. Classifiers

Type	Pros	Cons
Linear Regression	• Simple to implement	• Not guaranteed to work • Only supports binary labels
Logistic Regression	• Highly accurate • Flexible ways to regularize model for custom adjustment • Model responses are measures of probability	• Only supports binary labels

	• Easy to update model with new data	
Softmax Regression	• Supports multiclass classification • Model responses are measures of probability	• More complicated to implement

Linear regression is the easiest to implement because we've already done most of the hard work last chapter, but as you'll see, it's a terrible classifier. A much better classifier is the logistic regression algorithm. As the name suggests, it'll use logarithmic properties to define a better cost function. And lastly, softmax regression is a direct approach to solve multiclass classification. It is a natural generalization of logistic regression. It's called softmax regression because there's a function called *softmax* which is applied as the very last step.

4.1 Formal Notation

In mathematical notation, a classifier is a function $y = f(x)$, where x is the input data item and y is the output category (figure 4.1). Adopted from traditional scientific literature, we often refer to the input vector x as the *independent variable*, and the output y as the *dependent variable*.

Figure 4.1 A classifier produces discrete outputs, but may take either continuous or discrete inputs.

Formally, a category label is restricted to a range of values, just like an enum in Python. When the input features are not continuous values, we need to ensure our model can understand how to handle it. Usually, the set of functions in a model deal with continuous real numbers. If our data doesn't follow that assumption, then we'll need to preprocess the dataset to account for discrete variables, which falls in one of two types: ordinal or nominal (figure 4.2).

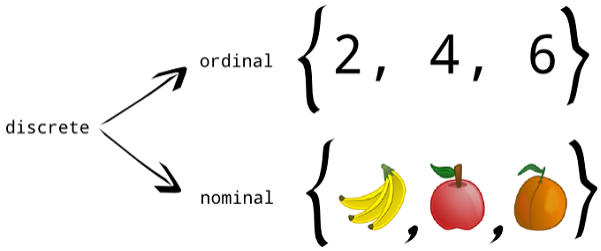


Figure 4.2 There are two types of discrete sets, those that can be ordered (ordinal) and those that cannot (nominal).

Values of an ordinal type, as the name suggests, can be ordered. For example, the values in a set of even numbers from *1* to *10* are ordinal because integers can be compared with each other. On the other hand, an element from a set of fruits *{banana, apple, orange}* might not come with a natural ordering. We call values from such a set nominal, because they can only be described by their names.

A simple approach to represent nominal variables in a dataset is to assign a number to each label. Our set {banana, apple, orange} could instead be processed as *{0, 1, 2}*. However, some classification models may have a strong bias about how the data behaves. For example, linear regression would interpret our apple as mid-way between a banana and an orange, which makes no natural sense.

A simple work-around to represent nominal categories of a dependent variable is by adding what are called *dummy variables* for each value of the nominal variable. In this example, the "fruit" variable would be removed, and replaced by three separate variables: "banana," "apple," and "orange." Each variable holds a value of *0* or *1* (figure 4.3), depending on whether the category for that fruit holds true.

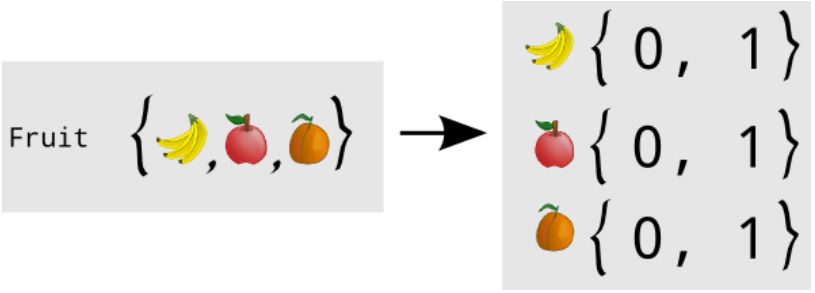


Figure 4.3 If the values of a variable are discrete, then they might need to be preprocessed. One solution is to treat each of the discrete values as a Boolean variable, as shown on the right. Banana, apple, and orange are 3 newly added variables, each having values 0 or 1. The original "fruit" variable is removed.

Just like in linear regression from chapter 3, the learning algorithm must traverse the possible functions supported by the underlying model, called *M*. In linear regression, the model was parameterized by *w*. The function *y = M(w)* can then be tried out to measure its cost. In the end, we choose a value of *w* with the least cost. The only difference between regression and classification is that the output is no longer a continuous spectrum, but instead a discrete set of class labels.

> **EXERCISE 4.1** Is it a better idea to treat each of the following as a regression or classification task: (a) predicting stock prices, (b) deciding whether to buy or sell a stock, (c) rating the quality of a computer on a 1-10 scale
>
> **ANSWER** (a) regression, (b) classification, (c) either

Since the input/output types for regression are even more general than that of classification, nothing prevents you from running a linear regression algorithm on a classification task. In fact, that's exactly what we'll do in section 4.3. Before we begin implementing TensorFlow code, it's important to gauge the strength of a classifier. The next section covers state-of-the-art approaches of measuring a classifier's success.

4.2 Measuring Performance

Before you begin writing classification algorithms, you should be able to check the success of your results. This section will cover some essential techniques to measure performance in classification problems.

4.2.1 Accuracy

Do you remember those multiple-choice exams in grad-school or university? Classification problems in machine learning are very similar. Given a statement, your job is to classify it as one of the given multiple-choice "answers". If you only have two choices, such as in a true or false exam, then we call it a *binary classifier*. If this were a graded exam in school, the typical way to measure your score would be to count the number of correct answers and divide it by the total number of questions.

Machine learning adopts this same scoring strategy and calls it *accuracy*. Accuracy is measured by the following formula.

```
accuracy = #correct / #total
```

This formula gives a crude summary of the performance which may be sufficient if you're only worried about the overall correctness of the algorithm. However, the accuracy measure doesn't reveal a breakdown of correct and incorrect results per each label.

To account for this limitation, a *confusion matrix* is a more detailed report of a classifier's success. A useful way to describe how well a classifier performs is by inspecting how it performs on each of the classes.

For instance, consider a binary classifier with a "positive" and "negative" label. As shown in figure 4.4, a confusion matrix is a table that compares how the predicted responses compare with actual ones. Data items that are correctly predicted as positive are called *true-positives* (TP). Those that are incorrectly predicted as positive are called *false positives* (FP). If the algorithm accidentally predicts an element to be negative when in reality it is positive, we call this situation a *false negative (FN)*. Lastly, when the prediction and reality both agree that a data item is a negative label, it's called a *true negative (TN)*.

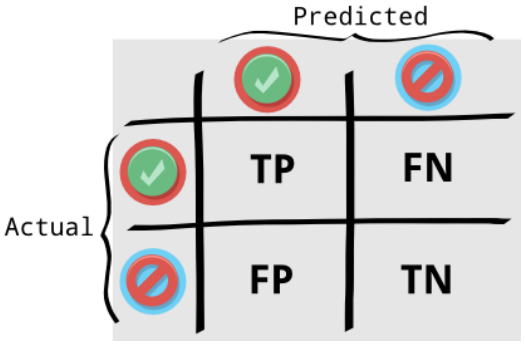


Figure 4.4 We can compare predicted results to actual results using a matrix of positive (green check mark) and negative (red cross) labels.

4.2.2 Precision and Recall

Although the definitions of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) are all useful individually, the true power comes in the interplay between them.

The ratio of true positives to total positive examples is called *precision*. It is a score of how likely a positive prediction is correct. The left column in figure 4.4 is the total number of positive predictions (TP + FP), so the equation for precision is the following.

```
precision = TP / (TP + FP)
```

The ratio of true positives to all possible positives is called *recall*. It measures the ratio of true positives found. In other words, it is a score of how many true-positives were successfully predicted (that is, "recalled"). The top row in figure 4.4 is the total number of all positives (TP + FN), so the equation for recall is the following.

```
recall = TP / (TP + FN)
```

Simply put, precision is the portion of your predictions you got right and recall is proportion of the right things you identified in the final set.

4.2.3 Receiver operating characteristic (ROC) curve

Because binary classifiers are among the most popular tools, many mature techniques exist for measuring their performance, such as the *ROC curve*. The ROC curve is a plot of trade-offs between false-positives and true-positives. The x-axis is the measure of false-positive values, and the y-axis is the measure of true-positive values.

A binary classifier reduces its input feature vector into a number and then decides the class based on whether the number is greater than or less than a specified threshold. As we adjust a threshold of the machine learning classifier, we plot the various values of false-positive and true-positive rates.

A robust way to compare various classifiers is by comparing their ROC curves. When two curves don't intersect, then one method is certainly better than the other. See figure 4.5 for an example.

EXERCISE 4.1 How would a 100% correct rate (all true positives, no false positives) look like as a point on a ROC curve?

ANSWER The point for a 100% correct rate would be located on the positive y-axis of the ROC curve.

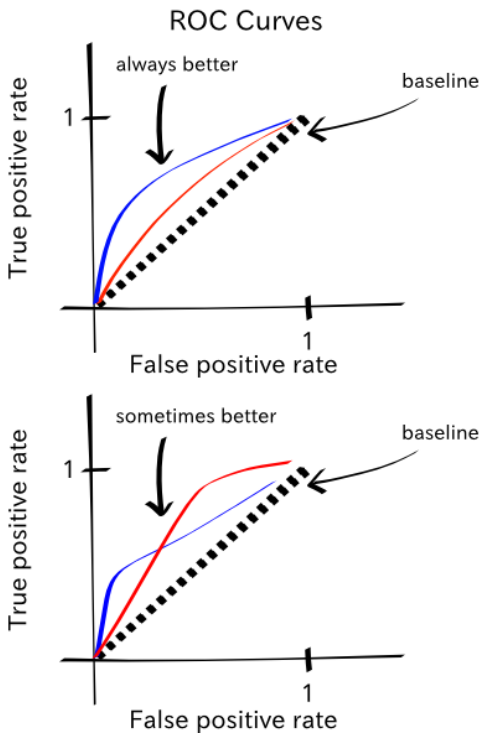


Figure 4.5 The principled way to compare different algorithms is by examining their ROC curves. When

the true positive rate is greater than the false positive rate at every situation, then it's straightforward to declare that one algorithm is dominant in terms of its performance. If the true-positive rate is less than the false-positive rate, the plot dips below the baseline shown by a dotted-line. Good algorithms are above the baseline, otherwise they do worse than random guessing.

4.3 Using linear regression for classification

One of the simplest ways to implement a classifier is to tweak a linear regression algorithm, like the ones in chapter 3. As a reminder, the linear regression model was a set of functions that look linear, $f(x) = wx$. The function $f(x)$ takes continuous real numbers as input and produces continuous real numbers as output. Remember, classification is all about discrete outputs. So, one way to force the regression model to produce a two-valued (in other words, binary) output is by setting values above some threshold to a number (such as 1) and values below that threshold to a different number (such as 0).

We proceed with the following motivating example. Imagine Alice is an avid chess player, and we have records of her win/loss history. Moreover, each game has a time-limit ranging between 1 and 10 minutes. We can plot the outcome of each game as shown in figure 4.6. The x-axis represents the time-limit of the game, and the y-axis signifies whether she won ($y = 1$) or lost ($y = 0$).

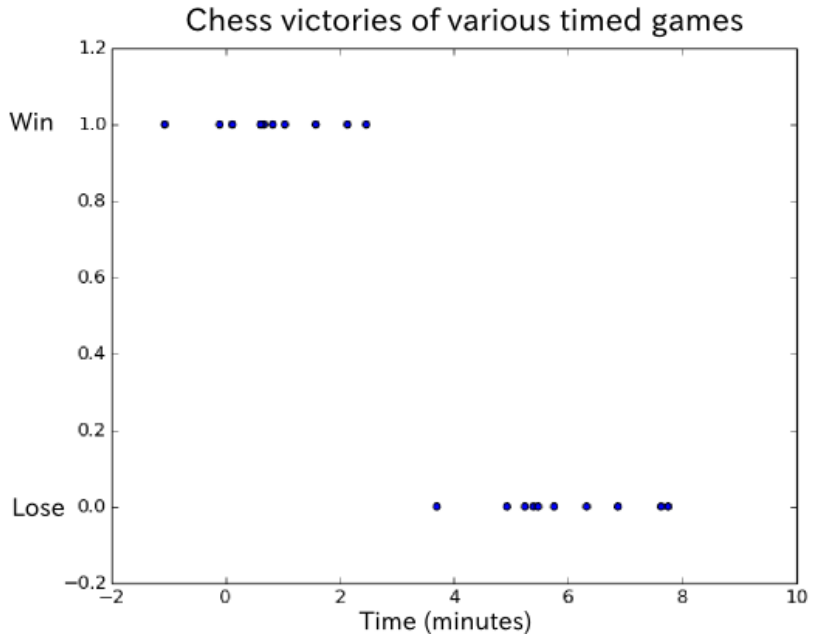


Figure 4.6 A visualization of a binary classification training dataset. The values are divided into two classes: all points where y = 1, and all points where y = 0.

As you see from the data, Alice is a quick thinker because she always wins short games. However, she usually loses games that have longer time limits. From the plot, we would like to predict the critical game time-limit that decides whether or not she'll win.

We want to challenge her to a game that we're sure of winning. If we choose an obvious long game such as one that takes 10 minutes, she'll refuse to play. So let's set up the game time to be as short as possible so she'll be willing to play against us, while tilting the balance to our advantage. 0

A linear fit on the data gives us something to work with. Figure 4.7 shows the best fit line computed using linear regression from listing 4.1. The value of the line is closer to 1 than it is to 0 for games that she will likely win. It appears that if we pick a time corresponding to when the value of the line is less than 0.5 (that is, when Alice is more likely to lose than to win), then we have a good chance of winning.

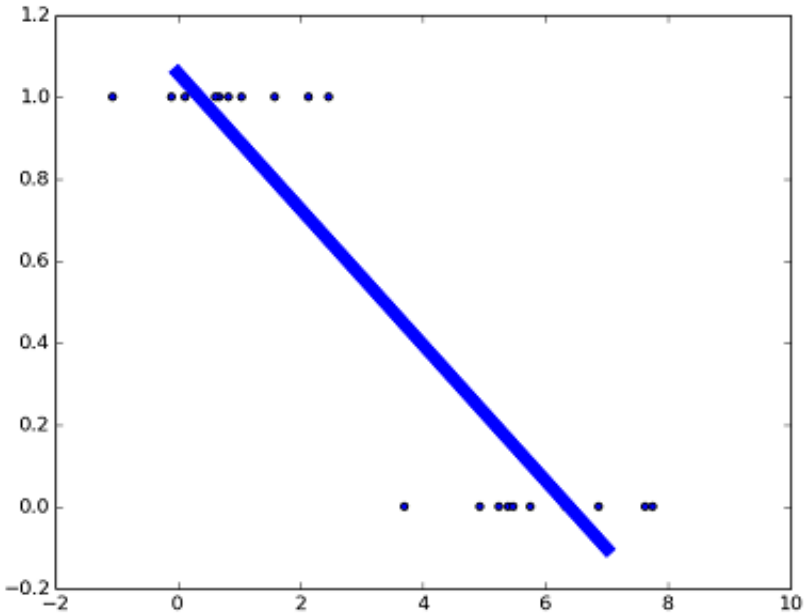


Figure 4.7 The diagonal line is the best fit line on a classification dataset. Clearly the line doesn't fit the data well, but it provides an out-of-date approach to classify new data.

The line is trying to fit the data as best as possible. By the nature of the training data, the model will respond with values near 1 for positive examples, and values near 0 for negative examples. Because we're modeling this data with a line, some input may produce values between 0 and 1. As you may imagine, values too far into one category will result in values greater than 1 or less than 0. We need a way to decide when an item belongs to one category over another. Typically, we choose the midpoint 0.5 as a deciding boundary (also called threshold).

Let's write our first classifer! Open a new Python source file, and call it `linear.py`. Follow the code in listing 4.1 to write the code. You'll be using linear regression to perform classification. In the TensorFlow code, you'll need to first define placeholder nodes and then inject values into them from the `session.run()` statement.

Listing 4.1 Using linear regression for classification

```
import tensorflow as tf          (1)
import numpy as np               (1)
import matplotlib.pyplot as plt  (1)

x_label0 = np.random.normal(5, 1, 10)  (2)
x_label1 = np.random.normal(2, 1, 10)  (2)
xs = np.append(x_label0, x_label1)     (2)
labels = [0.] * len(x_label0) + [1.] * len(x_label1)  (3)

plt.scatter(xs, labels)          (4)

learning_rate = 0.001            (5)
training_epochs = 1000           (5)

X = tf.placeholder("float")      (6)
Y = tf.placeholder("float")      (6)

def model(X, w):                 (7)
    return tf.add(tf.mul(w[1], tf.pow(X, 1)),  (7)
                  tf.mul(w[0], tf.pow(X, 0)))  (7)

w = tf.Variable([0., 0.], name="parameters")   (8)
y_model = model(X, w)            (9)
cost = tf.reduce_sum(tf.square(Y-y_model))     (10)

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)  (11)
```

(1) Import TensorFlow for the core learning algorithm, NumPy for manipulating data, and Matplotlib for visualizing
(2) Initialize fake data, 10 instances of each label
(3) Initialize the corresponding labels
(4) Plot the data
(5) Declare the hyper-parameters
(6) Set up the placeholder nodes for the input/output pairs
(7) Define a linear y = w1 * x + w0 model.
(8) Set up the parameter variables
(9) Define a helper variable because we'll refer to this multiple times
(10) Define the cost function
(11) Define the rule to learn the parameters

After designing the TensorFlow graph, listing 4.2 shows you how to open a new session and execute the graph. The `train_op` updates the model's parameters to better and better guesses. We run the `train_op` multiple times in a loop since each step iteratively improves the parameter estimate. Listing 4.2 generate a plot similar to figure 4.7 above.

Listing 4.2 Executing the graph

```
sess = tf.Session()  ⑫
init = tf.initialize_all_variables()   ⑫
sess.run(init)       ⑫

for epoch in range(training_epochs):   ⑬
    sess.run(train_op, feed_dict={X: xs, Y: labels})            ⑬
    current_cost = sess.run(cost, feed_dict={X: xs, Y: labels})  ⑭
    if epoch % 10 == 0:
        print(epoch, current_cost)     ⑮

w_val = sess.run(w)  ⑯
print('learned parameters', w_val)     ⑯

sess.close()         ⑰

all_xs = np.linspace(0, 10, 100)       ⑱
plt.plot(all_xs, all_xs*w_val[1] + w_val[0])  ⑱
plt.show()           ⑱
```

⑫ Open a new session and initialize the variables
⑬ Run the learning operation multiple times
⑭ Record the cost computed with the current parameters
⑮ Print out log info while the code runs
⑯ Print the learned parameters
⑰ Close the session when no longer in use
⑱ Show the best fit line

To measure success, we can count the number of correct predictions and compute a success rate. In listing 4.3, you will add two more nodes to your previous code in `linear.py` called `correct_prediction` and `accuracy`. You can then print the value of accuracy to see the success rate. The code can be executed right before closing the session.

Listing 4.3 Measuring accuracy

```
correct_prediction = tf.equal(Y, tf.to_float(tf.greater(y_model, 0.5)))  ①
accuracy = tf.reduce_mean(tf.to_float(correct_prediction))               ②

print('accuracy', sess.run(accuracy, feed_dict={X: xs, Y: labels}))      ③
```

① When the model's response is greater than 0.5, it should be a positive label, and vice versa
② Compute the percent of success
③ Print the success measure from provided input

The above listing 4.3 produces the following output:

```
('learned parameters', array([ 1.2816, -0.2171], dtype=float32))
('accuracy', 0.95)
```

If classification were that easy, this chapter would be over by now. Unfortunately, the linear regression approach fails miserably if we train on more extreme data, also called *outliers*.

For example, let's say Alice lost a game that took 20 minutes. We train the classifier on a dataset that includes this new outlier datapoint. The code in listing 4.4 simply replaces one of the game-times with the value of 20. Let's see how introducing an outlier affects the classifier's performance.

Listing 4.4 Linear regression failing miserably for classification

```
x_label0 = np.append(np.random.normal(5, 1, 9), 20)
```

When you re-run the code with the changes made in listing 4.3, you will see a result similar to figure 4.8.

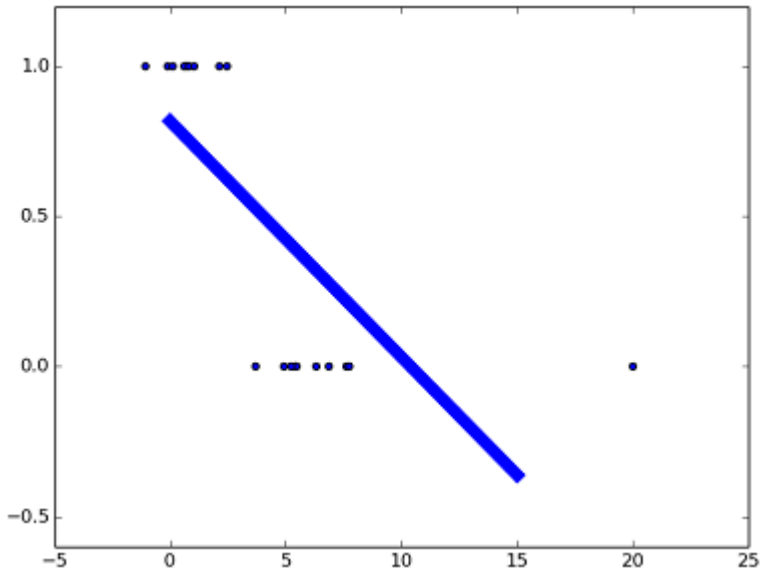


Figure 4.8 A new training element of value 20 greatly influences the best-fit line. The line is too sensitive to outlying data, and therefore linear regression is a sloppy classifier.

The original classifier suggested that we could beat Alice in a three-minute game. She'd probably agree to play such a short game. But with the revised classifier, if we stick with the same 0.5 threshold, is now suggesting that the shortest game she'll lose is five minutes. She'll likely refuse to play such a long game!

4.4 Using logistic regression

Logistic regression provides us with an analytic function with theoretical guarantees on accuracy and performance. It's just like linear regression, except we use a different cost function and slightly transform the model response function.

Let's revisit the linear function below.

```
linear(x) = w * x
```

In linear regression, a line with non-zero slope may range from negative infinity to infinity. If the only sensible results for classification are *0* or *1*, then it would be intuitive to instead fit a function with that property. Fortunately, the sigmoid function visualized in figure 4.9 works well because it converges to 0 or 1 very quickly.

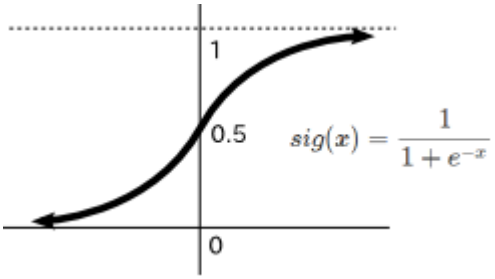


Figure 4.9 A visualization of the sigmoid function.

When x is *0*, the sigmoid function results in *0.5*. As x increases, the function converges to *1*. And as x decrease to negative infinity, the function converges to *0*.

In logistic regression our model is *sig(linear(x))*. Turns out, the best-fit parameters of this function implies a linear separation between the two classes. This separating line is also called a *linear decision boundary*.

4.4.1 Solving one-dimensional logistic regression

The cost function used in logistic regression is a bit different. Although we could just use the same cost function as before, it won't be as fast nor guarantee an optimal solution. The sigmoid function is the culprit here, because it causes the cost function to have many "bumps." TensorFlow and most other machine learning libraries work best with simple cost functions. Scholars have found a neat way to modify the cost function to use sigmoids for logistic regression.

The new cost function between the actual value y and model response h will be the two-part equation as follows.

```
Cost = -log( h )       if y = 1
Cost = -log( 1 - h )   if y = 0
```

We can condense the two equations into one long equation as follows.

```
Cost = -y log( h ) - (1 - y) log( 1 - h )
```

This function has exactly the qualities needed for efficient and optimal learning. Specifically, it's convex, but don't worry too much about what that means. We're trying to minimize the cost: think of cost as an altitude and the cost function as a terrain. We're trying to find the

lowest point in the terrain. It's a lot easier to find the lowest point in the terrain if there is no place you can ever go uphill. Such a place is called "convex." There are no hills.

You can think of it like a ball rolling down a hill. Eventually, the ball will settle to the bottom, which is the "optimal point." A non-convex function might have a rugged terrain, making it difficult to predict where a ball will roll. It might not even end up at the lowest point. Our function is convex, so the algorithm will easily figure out how to minimize this cost and "roll the ball downhill."

Convexity is nice, but correctness is also an important criterion when picking a cost function. How do we know this cost function does exactly what we indented it to do? To most intuitively answer that question, take a look at figure 4.10. Sure, figures are an informal way to convince you, but the technical discussion for why the cost function is optimal is beyond the scope of the book. If you're interested behind the mathematics of it, you'll be interested to learn that the cost function is derived from the principle of maximum entropy, which you can look up anywhere online.

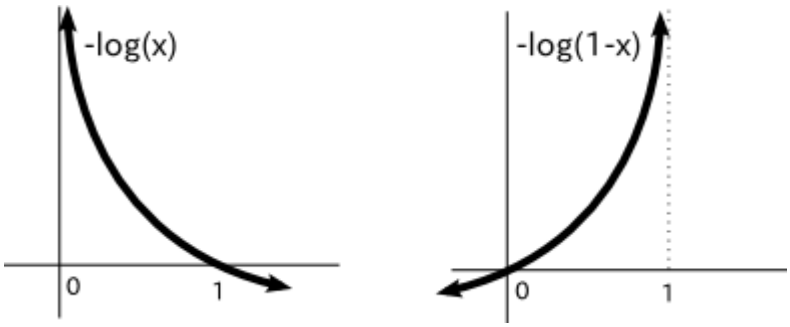


Figure 4.10 Here's a visualization of how the two different cost functions penalize values at 0 and 1. Notice how the left function heavily penalizes 0, but has no cost at 1. The right cost function displays the opposite phenomena.

See figure 4.10 for a best fit result from logistic regression on a one-dimensional dataset. The sigmoid curve that we will generate will provide a better linear decision boundary than that from linear regression.

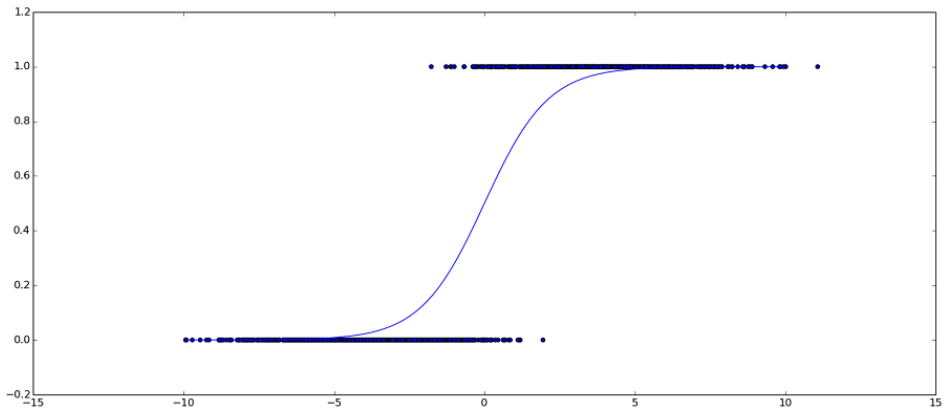


Figure 4.11 Here is a best fit sigmoid curve for a binary classification dataset. Notice how the curve resides within y = 0 and y = 1. That way, this curve is not that sensitive to outliers.

You'll start to notice a pattern in the code listings. In a simple/typical usage of TensorFlow, you generate some fake dataset, define placeholders, define variables, define a model, define a cost function on that model (which is often mean squared error or mean squared log error), you create a `train_op` using gradient descent, you iteratively feed it example data (possible with a label or output), and finally you collect the optimized values. Create a new source file called `logistic_1d.py` and follow along with listing 4.5, which will generate figure 4.11.

Listing 4.5 Using one-dimensional logistic regression

```
import numpy as np          (1)
import tensorflow as tf     (1)
import matplotlib.pyplot as plt     (1)

learning_rate = 0.01        (2)
training_epochs = 1000      (2)

def sigmoid(x):             (3)
    return 1. / (1. + np.exp(-x))    (3)

x1 = np.random.normal(-4, 2, 1000)   (4)
x2 = np.random.normal(4, 2, 1000)    (4)
xs = np.append(x1, x2)      (4)
ys = np.asarray([0.] * len(x1) + [1.] * len(x2))   (4)

plt.scatter(xs, ys)         (5)

X = tf.placeholder(tf.float32, shape=(None,), name="x")       (6)
Y = tf.placeholder(tf.float32, shape=(None,), name="y")       (6)
w = tf.Variable([0., 0.], name="parameter", trainable=True)   (7)
y_model = tf.sigmoid(-(w[1] * X + w[0]))   (8)
cost = tf.reduce_mean(-tf.log(y_model * Y + (1 - y_model) * (1 - Y)))        (9)

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)   (10)
```

```
with tf.Session() as sess:  ⓫
    sess.run(tf.initialize_all_variables())  ⓫
    prev_err = 0            ⓬
    for epoch in range(training_epochs):     ⓭
        err, _ = sess.run([cost, train_op], {X: xs, Y: ys})  ⓮
        print(epoch, err)
        if abs(prev_err - err) < 0.0001:     ⓯
            break
        prev_err = err      ⓰
    w_val = sess.run(w, {X: xs, Y: ys})      ⓱

all_xs = np.linspace(-10, 10, 100)           ⓲
plt.plot(all_xs, sigmoid(-(all_xs * w_val[1] + w_val[0])))   ⓲
plt.show()  ⓲
```

❶ Import relevant libraries
❷ Set the hyper-parameters
❸ Define a helper function to calculate the sigmoid function
❹ Initialize fake data
❺ Visualize the data
❻ Define the input/output placeholders
❼ Define the parameter node
❽ Define the model using TensorFlow's sigmoid function
❾ Define the cross-entropy loss function
❿ Define the minimizer to use
⓫ Open a session and define all variables
⓬ Define a variable to track of the previous error
⓭ Iterate until convergence or until maximum number of epochs reached
⓮ Computer the cost as well as update the learning parameters
⓯ Check for convergence
⓰ Update the previous error value
⓱ Obtain the learned parameter value
⓲ Plot the learned sigmoid function

And there you have it! If you were playing chess againt Alice, you now have a binary classifier to decide the threshold for when a chess match might result in a win or loss.

4.4.2 Solving two-dimensional logistic regression

Now we will explore how to use logistic regression with multiple independent variables. The number of independent variables corresponds to the number of dimensions. In our case, a two-dimensional logistic regression problem will try to label a pair of independent variables. The concepts learned in this section extrapolate to arbitrary dimensions.

> **MORE THAN TWO DIMENSIONS?** Let's say we're thinking about buying a new phone. The only attributes we care about are (1) operating system, (2) size, and (3) cost. Dear reader, I know you've acquired a richer taste, but for simplicity we're only interested in the three. The goal is to decide whether a phone is a worthwhile purchase. In this case there are three independent variables (the attributes of the phone), and one dependent variable (whether or not it's worth buying). So we regard this as a classification problem where the input vector is three-dimensional.

Consider the dataset shown in figure 4.12. It represents crime activity of two different gangs in a city. The first dimension is the x-axis, which can be thought of as the latitude, and the second dimension is the y-axis representing longitude. There's one cluster around (3, 2) and the other around (7, 6). Your job is to decide which gang is most likely responsible for a new crime that occurred on location (6,4).

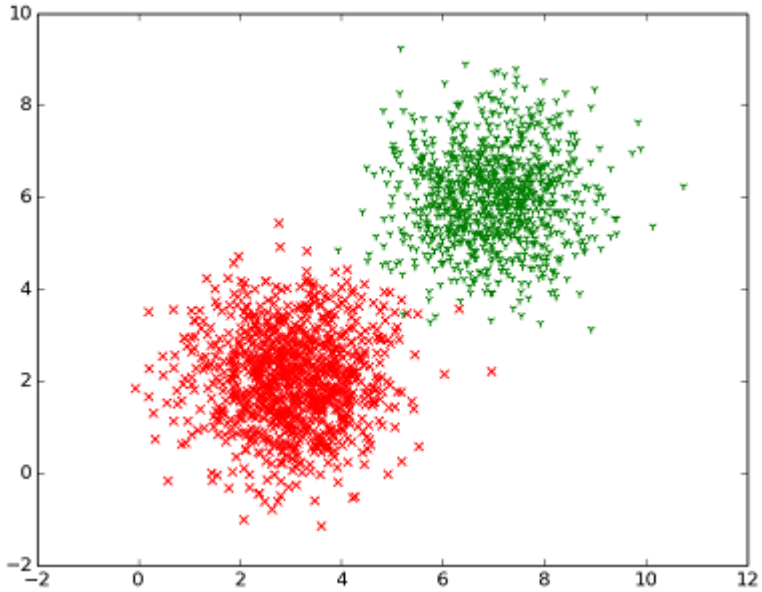


Figure 4.12 The x- and y-axis represent the two independent variables. The dependent variable holds two possible labels, represented by the shape and color of the plotted points.

Create a new source file called `logistic_2d.py` and follow along with listing 4.6.

Listing 4.6 Using two-dimensional logistic regression

```
import numpy as np          (1)
import tensorflow as tf     (1)
import matplotlib.pyplot as plt     (1)

learning_rate = 0.1         (2)
training_epochs = 2000      (2)

def sigmoid(x):             (3)
    return 1. / (1. + np.exp(-x))   (3)

x1_label1 = np.random.normal(3, 1, 1000)   (4)
x2_label1 = np.random.normal(2, 1, 1000)   (4)
x1_label2 = np.random.normal(7, 1, 1000)   (4)
x2_label2 = np.random.normal(6, 1, 1000)   (4)
x1s = np.append(x1_label1, x1_label2)      (4)
x2s = np.append(x2_label1, x2_label2)      (4)
```

```
ys = np.asarray([0.] * len(x1_label1) + [1.] * len(x1_label2))   ❹

X1 = tf.placeholder(tf.float32, shape=(None,), name="x1")        ❺
X2 = tf.placeholder(tf.float32, shape=(None,), name="x2")        ❺
Y = tf.placeholder(tf.float32, shape=(None,), name="y")          ❺
w = tf.Variable([0., 0., 0.], name="w", trainable=True)          ❻

y_model = tf.sigmoid(-(w[2] * X2 + w[1] * X1 + w[0]))            ❼
cost = tf.reduce_mean(-tf.log(y_model * Y + (1 - y_model) * (1 - Y)))  ❽
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)  ❽

with tf.Session() as sess:  ❾
    sess.run(tf.initialize_all_variables())  ❾
    prev_err = 0             ❾
    for epoch in range(training_epochs):     ❾
        err, _ = sess.run([cost, train_op], {X1: x1s, X2: x2s, Y: ys})       ❾
        print(epoch, err)    ❾
        if abs(prev_err - err) < 0.0001:     ❾
            break            ❾
        prev_err = err       ❾

    w_val = sess.run(w, {X1: x1s, X2: x2s, Y: ys})  ❿

x1_boundary, x2_boundary = [], []            ⓫
for x1_test in np.linspace(0, 10, 100):      ⓬
    for x2_test in np.linspace(0, 10, 100):  ⓬
        z = sigmoid(-x2_test*w_val[2] - x1_test*w_val[1] - w_val[0])  ⓭
        if abs(z - 0.5) < 0.01:              ⓭
            x1_boundary.append(x1_test)      ⓭
            x2_boundary.append(x2_test)      ⓭

plt.scatter(x1_boundary, x2_boundary, c='b', marker='o', s=20)  ⓮
plt.scatter(x1_label1, x2_label1, c='r', marker='x', s=20)      ⓮
plt.scatter(x1_label2, x2_label2, c='g', marker='1', s=20)      ⓮

plt.show()  ⓮
```

❶ Import relevant libraries
❷ Set the hyper-parameters
❸ Define a helper sigmoid function
❹ Initialize some fake data
❺ Define the input/output placeholder nodes
❻ Define the parameter node
❼ Define the sigmoid model using both input variables
❽ Define the learning step
❾ Create a new session, initialize variables, and learn parameters until convergence
❿ Obtain the learn parameter value before closing the session
⓫ Define arrays to hold boundary points
⓬ Loop through a window of points
⓭ If the model response is close the 0.5, then update the boundary points
⓮ Show the boundary line along with the data

Figure 4.13 visualizes the linear boundary line learned from the training data. A crime that occurs on this line has an equal chance of being part of either gang.

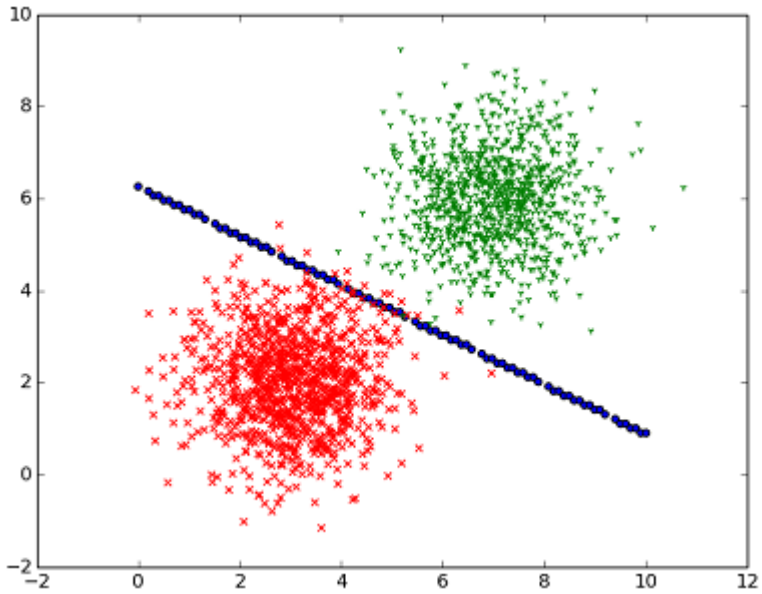


Figure 4.13 The diagonal dotted line represents when the probability between the two decisions is split equal. The confidence of making a decision increase as data lies further away from the line.

4.5 Multiclass classifier

So far, we've dealt with multidimensional input, but not multivariate output as shown in figure 4.14. For example, instead of binary labels on the data, what if we have 3, or 4, or 100 classes? Logistic regression requires two labels, no more.

Image classification, for example, is a popular multivariate classification problem because the goal is to decide the class of an image from a collection of candidates. A photograph may be bucketed into one of hundreds of categories.

To handle more than two labels, we may reuse logistic regression in a clever way (using a one-versus-all or one-versus-one approach) or develop a new approach (softmax regression). Let's look at each of the approaches in the next sections. The first two approaches require a decent amount of ad-hoc engineering, so we'll focus our efforts on softmax regression.

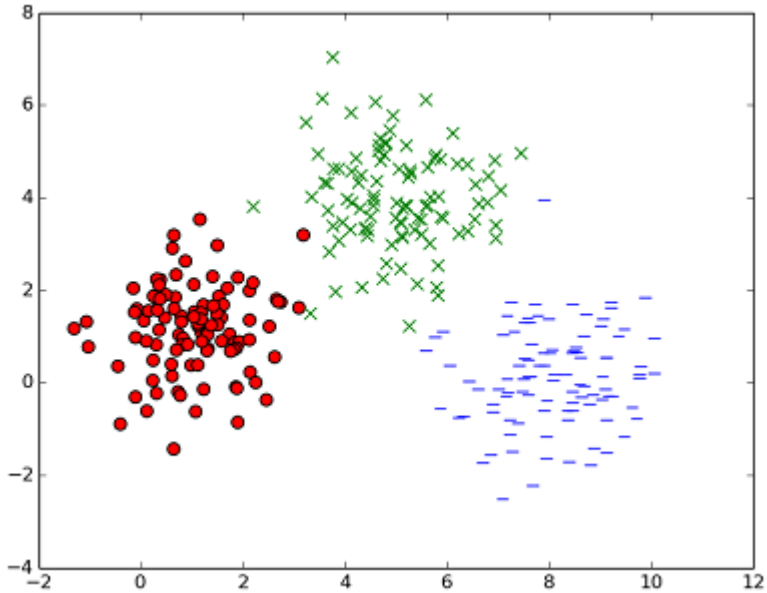


Figure 4.14 The independent variable is two-dimensional, indicated by the x and y-axis. The dependent variable can be one of three labels, shown by the color and shape of the datapoints.

4.5.1 One versus all

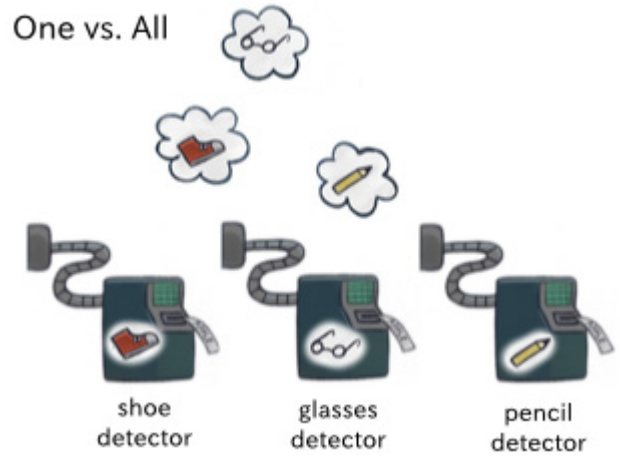


Figure 4.15 One versus All is a multi-class classifier approach that requires a detector for each class.

First we train a classifier for each of the labels as visualized in figure 4.15. If there are three labels, we have three classifier available to use: *f1*, *f2*, and *f3*. To test on new data, we run

each of the classifiers to see which one produced the most confident response. Intuitively, we label the new point by the label of the classifier that responded most confidently.

4.5.2 One versus one

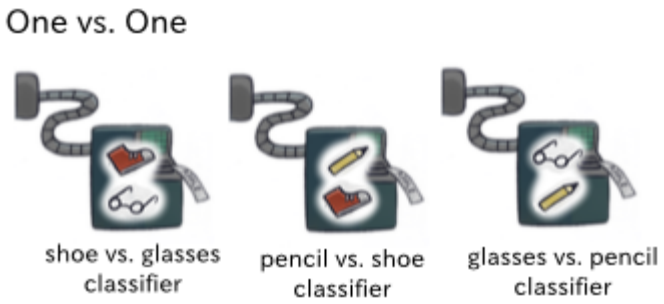


Figure 4.16 In One vs. One multi-class classification, there's a detector for each pair of classes.

First we train a classifier for each pair of labels (see figure 4.16). If there are three labels, then that's just three unique pairs. But for *k* number of labels, that's *k(k-1)/2* pairs of labels. On new data, we run all the classifiers and choose the class with the most wins.

4.5.3 Softmax regression

Softmax is named after the traditional max function, which takes a vector and returns the max value; however, softmax is not exactly the max function because it has the added benefit of being continuous and differentiable. As a result, it has the helpful properties for stochastic gradient descent to work efficiently.

In this type of multiclass classification setup, each class has a confidence (or probability) score for each input vector. The softmax step simply picks the highest scoring output.

Open a new file called `softmax.py` and follow along to listing 4.7 closely. First we will visualize some fake data to reproduce figure 4.14.

Listing 4.7 Visualizing multiclass data

```
import numpy as np                                  (1)
import matplotlib.pyplot as plt                     (1)

x1_label0 = np.random.normal(1, 1, (100, 1))        (2)
x2_label0 = np.random.normal(1, 1, (100, 1))        (2)
x1_label1 = np.random.normal(5, 1, (100, 1))        (3)
x2_label1 = np.random.normal(4, 1, (100, 1))        (3)
x1_label2 = np.random.normal(8, 1, (100, 1))        (4)
x2_label2 = np.random.normal(0, 1, (100, 1))        (4)

plt.scatter(x1_label0, x2_label0, c='r', marker='o', s=60)   (5)
plt.scatter(x1_label1, x2_label1, c='g', marker='x', s=60)   (5)
plt.scatter(x1_label2, x2_label2, c='b', marker='_', s=60)   (5)
```

```
plt.show()                                    ❺
```

❶ Import NumPy and Matplotlib
❷ Generate points near (1, 1)
❸ Generate points near (5, 4)
❹ Generate points near (8, 0)
❺ Visualize the three labels on a scatter plot

Next, in listing 4.8 we will set up the training and test data to prepare for the softmax regression step. The labels must be represented as a vector where only one element is *1* and the rest are *0*s. This representation is called *one-hot encoding*. For instance, if there are three labels, they would be represented as the following vectors: `[1, 0, 0]`, `[0, 1, 0]`, and `[0, 0, 1]`.

> **EXERCISE 4.2** One-hot encoding might appear like an unnecessary step. Why not just have a 1-dimensional output where values of 1, 2, and 3 represent the three classes.
>
> **ANSWER** Regression may induce a semantic structure in the output. If outputs are similar, regression implies that their inputs were also similar. If we just use one dimension, then we're implying labels 2 and 3 are more similar to each other than 1 and 3. We must be careful about making unnecessary or incorrect assumptions, so it's a safe bet to use one-hot encoding.

Listing 4.8 Setting up training and test data for multiclass classification

```
xs_label0 = np.hstack((x1_label0, x2_label0))       ❶
xs_label1 = np.hstack((x1_label1, x2_label1))       ❶
xs_label2 = np.hstack((x1_label2, x2_label2))       ❶
xs = np.vstack((xs_label0, xs_label1, xs_label2))   ❶

labels = np.matrix([[1., 0., 0.]] * len(x1_label0) + [[0., 1., 0.]] * len(x1_label1) + [[0.,
       0., 1.]] * len(x1_label2))   ❷

arr = np.arange(xs.shape[0])                ❸
np.random.shuffle(arr)                      ❸
xs = xs[arr, :]                             ❸
labels = labels[arr, :]                     ❸

test_x1_label0 = np.random.normal(1, 1, (10, 1))    ❹
test_x2_label0 = np.random.normal(1, 1, (10, 1))    ❹
test_x1_label1 = np.random.normal(5, 1, (10, 1))    ❹
test_x2_label1 = np.random.normal(4, 1, (10, 1))    ❹
test_x1_label2 = np.random.normal(8, 1, (10, 1))    ❹
test_x2_label2 = np.random.normal(0, 1, (10, 1))    ❹
test_xs_label0 = np.hstack((test_x1_label0, test_x2_label0))   ❹
test_xs_label1 = np.hstack((test_x1_label1, test_x2_label1))   ❹
test_xs_label2 = np.hstack((test_x1_label2, test_x2_label2))   ❹

test_xs = np.vstack((test_xs_label0, test_xs_label1, test_xs_label2))   ❹
test_labels = np.matrix([[1., 0., 0.]] * 10 + [[0., 1., 0.]] * 10 + [[0., 0., 1.]] * 10)   ❹

train_size, num_features = xs.shape   ❺
```

1. Combine all input data into one big matrix
2. Create the corresponding one-hot labels
3. Shuffle the dataset
4. Construct the training dataset and labels
5. The shape of the dataset tells you the number examples and features per example

Finally, in listing 4.9 we will use softmax regression. Unlike the sigmoid function in logistic regression, here we will use the softmax function provided by the TensorFlow library. The softmax function is similar to the max function which simply outputs the maximum value from a list of numbers. It's called softmax because it's a "soft" or "smooth" approximation of the max function, which is not smooth or continuous (and that's bad).

Listing 4.9 Using softmax regression

```
import tensorflow as tf

learning_rate = 0.01    (2)
training_epochs = 1000  (2)
num_labels = 3          (2)
batch_size = 100        (2)

X = tf.placeholder("float", shape=[None, num_features])  (3)
Y = tf.placeholder("float", shape=[None, num_labels])    (3)

W = tf.Variable(tf.zeros([num_features, num_labels]))    (4)
b = tf.Variable(tf.zeros([num_labels]))        (4)
y_model = tf.nn.softmax(tf.matmul(X, W) + b)  (5)

cost = -tf.reduce_sum(Y * tf.log(y_model))    (6)
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)    (6)

correct_prediction = tf.equal(tf.argmax(y_model, 1), tf.argmax(Y, 1))         (7)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))               (7)
```

2. Define hypter-parameters
3. Define the input/output placeholder nodes
4. Define the model parameters
5. Design the softmax model
6. Set up the learning algorithm
7. Define an op to measure success rate

Now that you've defined the TensorFlow computation graph, execute it from a session. We'll try a new form of iteratively updating the parameters this time, called *batch learning*. Instead of passing in the data one at a time, we'll run the optimizer on batches of data. This speeds things up but introduces risk of converging to a local optimum solution instead of the global best. Follow listing 4.10 for running the optimizer in batches.

Listing 4.10 Executing the graph

```
with tf.Session() as sess:                (8)
    tf.initialize_all_variables().run()   (8)
```

```
    for step in range(training_epochs * train_size // batch_size):  ❾
        offset = (step * batch_size) % train_size            ❿
        batch_xs = xs[offset:(offset + batch_size), :]       ❿
        batch_labels = labels[offset:(offset + batch_size)]  ❿
        err, _ = sess.run([cost, train_op], feed_dict={X: batch_xs, Y: batch_labels})  ⓫
        print (step, err)  ⓬

    W_val = sess.run(W)    ⓭
    print('w', W_val)      ⓭
    b_val = sess.run(b)    ⓭
    print('b', b_val)      ⓭
    print "accuracy", accuracy.eval(feed_dict={X: test_xs, Y: test_labels})  ⓮
```

❽ Open a new session and initialize all variables
❾ Loop only enough times to complete a single pass through the dataset
❿ Retrieve a subset of the dataset corresponding to the current batch
⓫ Run the optimizer on this batch
⓬ Print on-going results
⓭ Print final learned parameters
⓮ Print success rate

The final output of running the softmax regression algorithm on our dataset result in the following:

```
('w', array([[-2.101, -0.021,  2.122],
             [-0.371,  2.229, -1.858]], dtype=float32))
('b', array([10.305, -2.612, -7.693], dtype=float32))
Accuracy 1.0
```

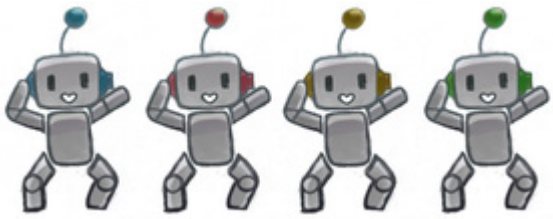
4.6 Summary

Because classification is a very useful machine learning technique, it has matured into a well-studied topic. Let's summarize what we've learned about it so far.

- There are many possible ways to solve classification problems, but logistic regression and softmax regression are some of the most robust in terms of accuracy and performance.
- It is important to preprocess data before running classification. For example, discrete independent variables can be readjusted into binary variables.
- So far, we approach classification from the point of view of regression. In later chapters, we will revisit classification using neural networks.
- There are various ways to approach multiclass classification. There's no clear answer to which one you should try first among one-vs-one, one-vs-all, and softmax regression. However, the softmax approach is a little more hands-free and it allows you to introduce regularization.

5

Automatically clustering data



This chapter covers

- Basic clustering using k-means
- Representing audio
- Audio segmentation
- Clustering with a self-organizing map

Suppose there's a collection of not-pirated-totally-legal mp3s on your hard drive. All your songs are crowded in one massive folder. But it might help to automatically group together similar songs and organize them into categories like "country," "rap," "rock," and so on. This act of assigning an item to a group (such as an mp3 to a playlist) in an unsupervised fashion is called *clustering*.

The previous chapter on classification assumes you're given a training dataset of correctly labeled data. Unfortunately, we don't always have that luxury when we collect data in the real-world. For example, suppose we would like to divide up a large amount of music into interesting playlists. How could we possibly group together songs if we don't have direct access to their metadata?

Spotify, SoundCloud, Google Music, Pandora, and many other music streaming services try to solve this problem to recommend similar songs to customers. Their approach includes a mixture of various machine learning techniques, but clustering is often at the heart of the solution.

The overall idea of clustering is that two items in the same cluster are "closer" to each other than items that belong to separate clusters. That is the general definition, leaving the interpretation of "closeness" open. For example, perhaps cheetahs and leopards belong in the same cluster, whereas elephants belong to another when closeness is measured by how similar two species are in the hierarchy of biological classification (family, genus, and species).

You can image there are many clustering algorithms out there. In this chapter we'll focus on two types, namely *k-means* and *self-organizing map*. These approaches are completely *unsupervised*, meaning they fit a model without ground-truth examples.

5.1 Traversing files in TensorFlow

Some common input types in machine learning algorithms are audio and image files. These data files have various implementations: an image can be encoded as a PNG or JPEG, and an audio file can be an MP3 or WAV. In this chapter, we will investigate how to read audio files as input to our clustering algorithm so we automatically group together music that sounds similar.

Reading files from disk isn't exactly a machine learning specific ability. You can use a variety of python libraries to load files onto memory, such as Numpy or Scipy. Some developers like to treat the data pre-processing step separately from the machine learning

step. There's no absolute right or wrong way to manage the pipeline, but we'll try to use TensorFlow for both the data pre-processing as well as the learning.

TensorFlow provides an operator to list files in a directory called `tf.train.match_filenames_once(…)`. We can then pass this information along to a queue operator `tf.train.string_input_producer(…)`. That way, we can access a filename one at a time, without loading everything at once. Given a filename, we can decode the file to retrieve usable data. Figure 5.1 outlines the whole process of using the queue.

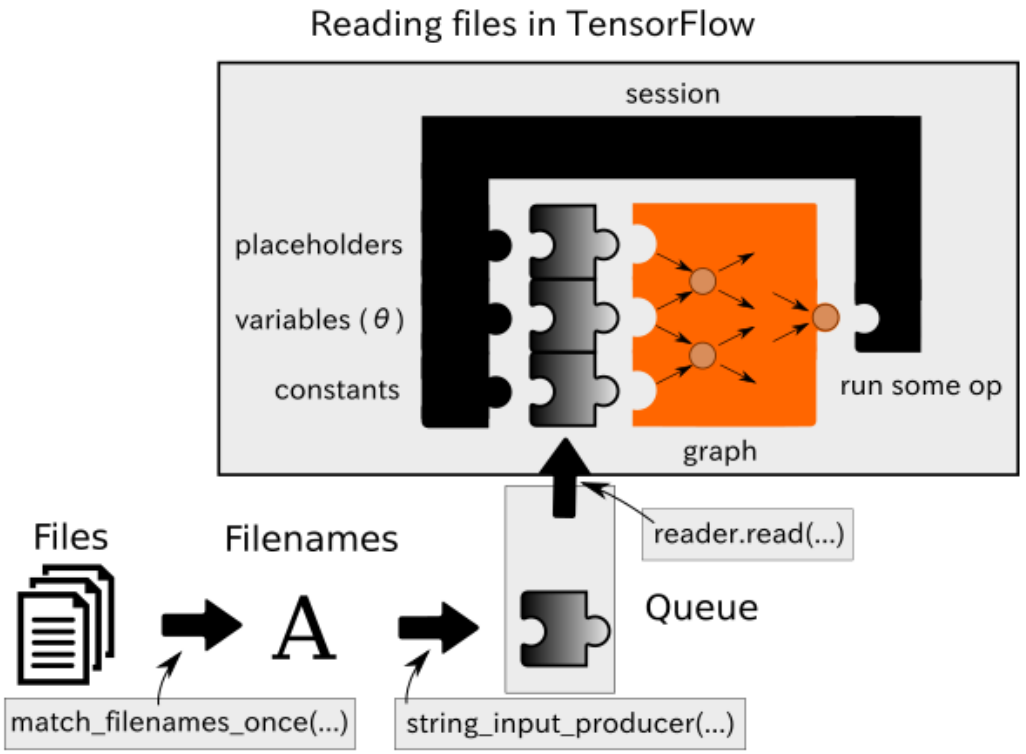


Figure 5.1 You can use a queue in TensorFlow to read files. The queue is built into the TensorFlow framework, and you can use the reader.read(...) function to access (and dequeue) it.

See listing 5.1 for how an implementation of how to read files from disk in TensorFlow.

Listing 5.1 Traversing a directory for data

```
import tensorflow as tf

filenames = tf.train.match_filenames_once('./audio_dataset/*.wav') ❶
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)          ❷
```

```
reader = tf.WholeFileReader()    ❸
filename, file_contents = reader.read(filename_queue)    ❹

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    num_files = sess.run(count_num_files)  ❺

    coord = tf.train.Coordinator()          ❻
    threads = tf.train.start_queue_runners(coord=coord)  ❻

    for i in range(num_files):  ❼
        audio_file = sess.run(filename)     ❼
        print(audio_file)       ❼
```

❶ Store filenames that match a pattern
❷ Set up an pipeline for retrieving filenames randomly
❸ Natively read a file in TensorFlow
❹ Run the reader to extract file data
❺ Count the number of files
❻ Initialize threads for the filename queue
❼ Loop through the data one by one

5.2 Extracting features from audio

Machine learning algorithms are typically designed to use feature vectors as input; however, sound files are a very different format. We need a way to extract features from sound files to create feature vectors.

It helps to understand how these files are represented. Our ears interpret audio from a series of vibrations through air. By recording the vibration properties, we can store sound in a data format. If you've ever seen a vinyl record, you've probably noticed the representation of audio as grooves indented in the disk.

The real world is continuous but computers store data in discrete values. The sound is digitalized into a discrete representation through an analog to digital converter (ADC). You can think about sound as a fluctuation of a wave over time. However, that data is too noisy and difficult to comprehend.

An equivalent way to represent a wave is by examining the frequencies that make it up at each time interval. This perspective is called the frequency domain. It's easy to convert between time domain and frequency domains using a mathematical operation called a discrete Fourier transform (commonly Fast Fourier transform). We will use this technique to extract a feature vector out of our sound.

There's a handy python library to view audio in this frequency domain. Download it from https://github.com/BinRoot/BregmanToolkit/archive/master.zip. Extract it, and then run the following command to set it up.

```
$ python setup.py install
```

A sound may produce 12 kinds of pitches. In music terminology, the 12 pitches are C, C#, D, D#, E, F, F#, G, G#, A, A#, and B. Listing 5.2 shows how to retrieve the contribution of each

pitch in a 0.1 second interval, resulting in a matrix with 12 rows. The number of columns grows as the length of the audio file increases. Specifically, there will be *10*t* columns for a *t* second audio. This matrix is also called a *chromogram* of the audio.

Listing 5.2 Representing audio in Python

```
from bregman.suite import *

def get_chromogram(audio_file):  ❶
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)  ❷
    return F.X  ❸
```

❶ Pass in the filename
❷ Use these parameters to describe 12 pitches every 0.1 seconds
❸ Represents the values of a 12-dimensional vector 10 times a second

The chromogram output will be a matrix visualized in figure 5.2. A sound clip can be read as a chromogram, and a chromogram is a recipe for generating a sound clip. Now we have a way to convert between audio and matrices. And as you have learned, most machine learning algorithms accept feature vectors as a valid form of data. That said, the first machine learning algorithm we'll look at is k-means clustering.

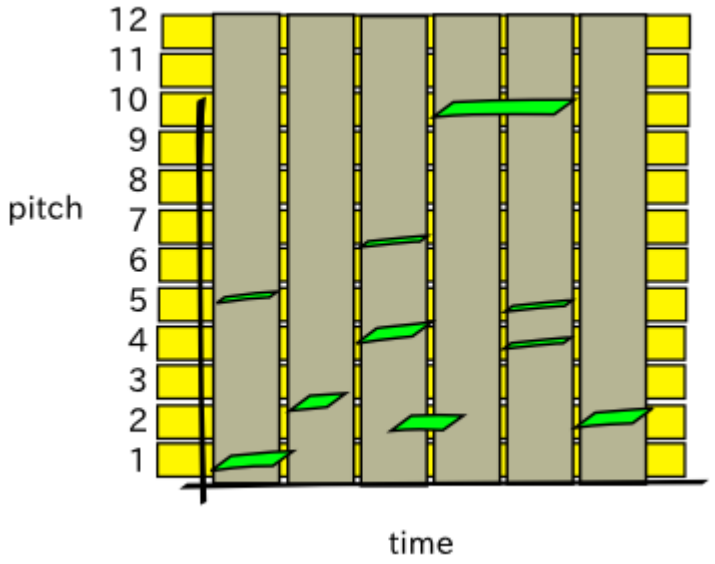


Figure 5.2 Visualization of the chromogram matrix where the x-axis represents time and the y-axis represents pitch class. The green markings indicate a presence of that pitch at that time.

To run machine learning algorithms on our chromogram, we first need to decide how we're going to represent a feature vector. One idea is to simplify the audio by only looking at the most significant pitch class per time interval, as shown in figure 5.3.

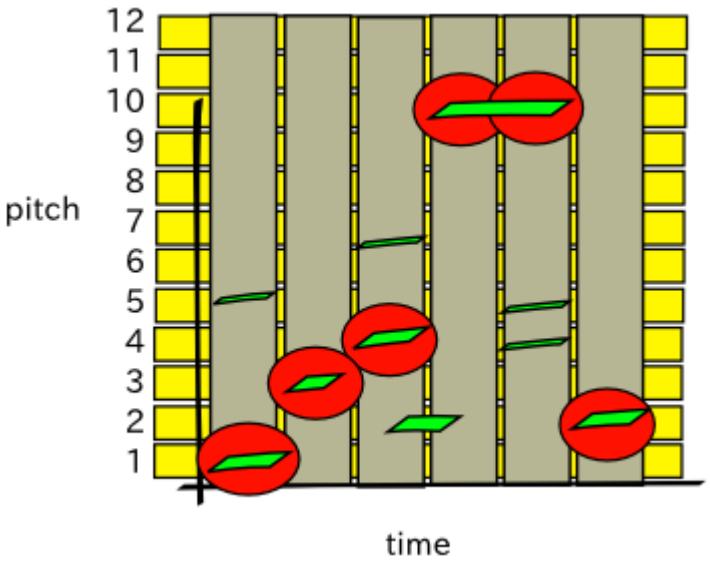


Figure 5.3 The most influential pitch at every time interval is highlighted. You can think of it as the loudest pitch at each time interval.

Then we count the number of times each pitch shows up in the audio file. Figure 5.4 shows this data as a histogram, forming a 12-dimensional vector. If we normalize the vector so that all the counts add up to 1, then we can easily compare audio of different lengths.

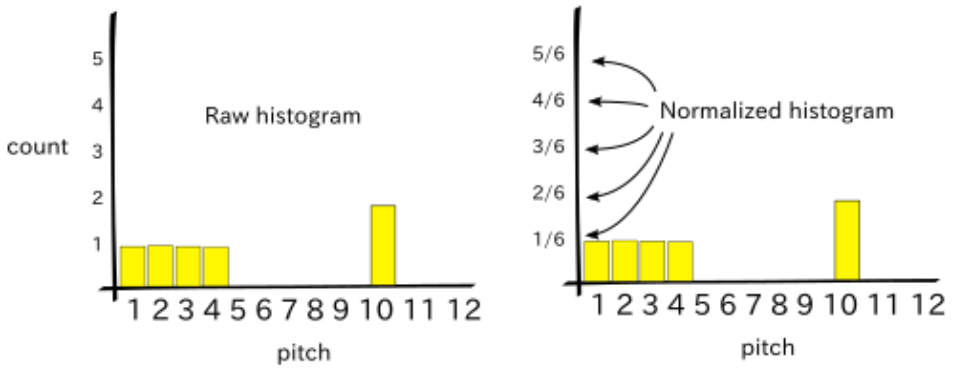


Figure 5.4 We count the frequency of loudest pitches heard at each interval to generate this histogram, which acts as our feature vector.

Let's take a look at listing 5.3 to generate this histogram.

Listing 5.3 Obtaining a dataset for k-means

```
import tensorflow as tf
import numpy as np
from bregman.suite import *


filenames = tf.train.match_filenames_once('./audio_dataset/*.wav')
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)
reader = tf.WholeFileReader()
filename, file_contents = reader.read(filename_queue)

chromo = tf.placeholder(tf.float32) (1)
max_freqs = tf.argmax(chromo, 0)    (1)

def get_next_chromogram(sess):
    audio_file = sess.run(filename)
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)
    return F.X

def extract_feature_vector(sess, chromo_data): (2)
    num_features, num_samples = np.shape(chromo_data)
    freq_vals = sess.run(max_freqs, feed_dict={chromo: chromo_data})
    hist, bins = np.histogram(freq_vals, bins=range(num_features + 1))
    return hist.astype(float) / num_samples

def get_dataset(sess): (3)
    num_files = sess.run(count_num_files)
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
    xs = []
    for i in range(num_files):
        chromo_data = get_next_chromogram(sess)
        x = [extract_feature_vector(sess, chromo_data)]
        x = np.matrix(x)
        if len(xs) == 0:
            xs = x
        else:
            xs = np.vstack((xs, x))
    return xs
```

(1) Create an op to identify the pitch with the biggest contribution
(2) Convert a chromogram into a feature vector
(3) Construct a matrix where each row is a data item

5.3 K-means clustering

The k-means algorithm is one of the oldest yet most robust ways to cluster data. The *k* in k-means is a variable representing a natural number. So you can imagine there's 3-means, or 4-means clustering. Thus, the first step of k-means clustering is to choose a value for *k*. Just to be more concrete, let's pick *k* = *3*. With that in mind, the goal of 3-means clustering is to divide the dataset into 3 categories (also called *clusters*).

Choosing the number of clusters

Choosing the right number of clusters often depends on the task. For example, suppose you're planning an event for hundreds of people, both young and old. If you have the budget for only two entertainment options, then you can use *k*-means clustering with *k = 2* to separate the guests into two age groups. Other times, it's not as obvious what the value of *k* should be. Automatically figuring out the value of *k* is a bit more complicated, so we won't touch on that much in this section.

The k-means algorithm treats data points as points in space. If our dataset is a collection of guests in an event, we can represent each one by his or her age. Thus, our dataset is a collection of feature vectors. In this case, each feature vector is only 1-dimensional, because we're only considering the age of the person.

For clustering music by the audio data, the data points are feature vectors from the audio files. If two points are close together, that means their audio features are similar. We want to discover which audio files belong in the same neighborhood, because those clusters will probably be a good way to organize our music files.

The midpoint of all the points in a cluster is called its *centroid*. Depending on the audio features we choose to extract, a centroid could capture concepts such as "loud sound," "high-pitched sound," or "saxophone-like sound." It's important to note that the k-means algorithm assigns non-descript labels, such as "cluster 1," "cluster 2," or "cluster 3." Figure 5.5 shows examples of the sound data.

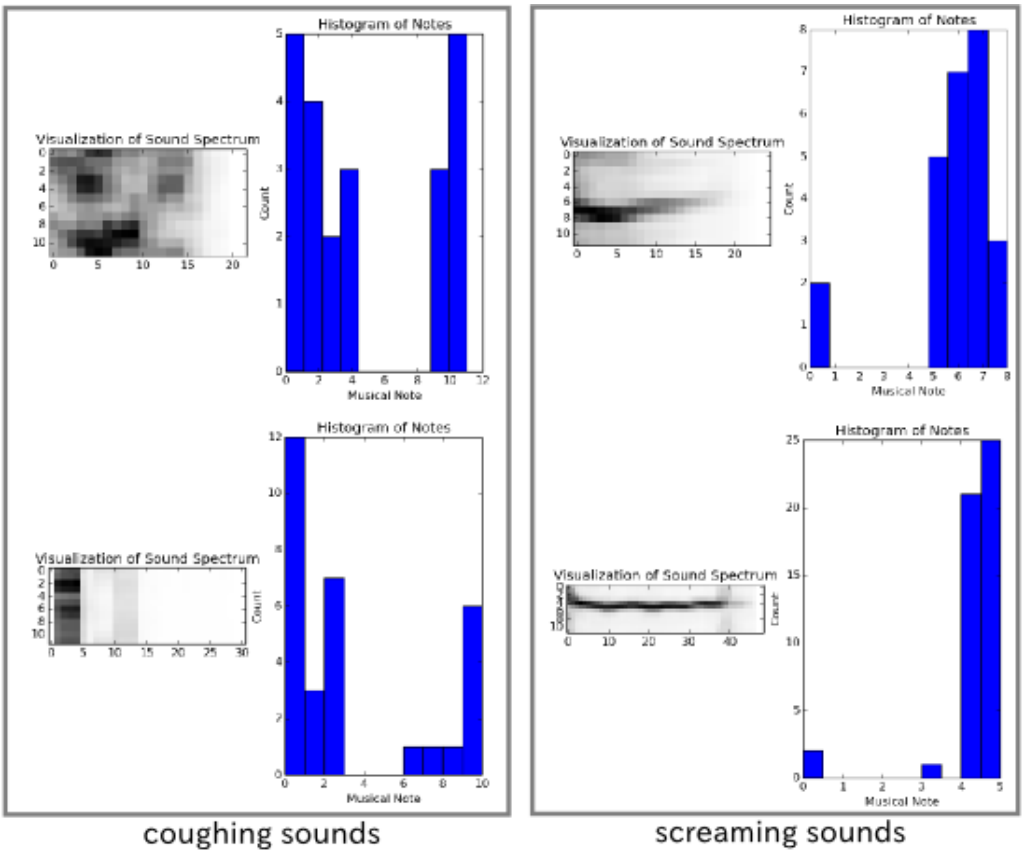


Figure 5.5 Here are four examples of audio files. As you can see, the two on the right appear to have similar histograms. The two on the left also have similar histograms. Our clustering algorithms will be able to group these sounds together.

The k-means algorithm assigns a feature vector to one of the *k* clusters by choosing the cluster whose centroid is closest to it. The k-means algorithm starts by guessing the cluster location. It iteratively improves its guess over time. The algorithm either converges when it no longer improves the guesses, or it stops after a maximum number of attempts.

The heart of the algorithm consists of two tasks: (1) assignment and (2) re-centering.

1. In the assignment step, we assign each data item (also called a feature vector) to a category of the closest centroid.
2. In the re-centering step, we calculate the midpoints of the newly updated clusters. These two steps repeat to provide better and better clustering results, and the algorithm stops when either it repeated a desired number of times or the assignments no longer change.

Listing 5.4 shows how to implement the k-means algorithm using the dataset generated by listing 5.3. For simplicity we'll choose k = 2, so that we can easily verify that our algorithm partitions the audio files into two dissimilar categories. We'll use the first k vectors as initial guesses for centroids.

Listing 5.4 Implementing k-means

```
k = 2                                    ❶
max_iterations = 100                     ❷

def initial_cluster_centroids(X, k):     ❸
    return X[0:k, :]

def assign_cluster(X, centroids):        ❹
    expanded_vectors = tf.expand_dims(X, 0)
    expanded_centroids = tf.expand_dims(centroids, 1)
    distances = tf.reduce_sum(tf.square(tf.sub(expanded_vectors, expanded_centroids)), 2)
    mins = tf.argmin(distances, 0)
    return mins

def recompute_centroids(X, Y):           ❺
    sums = tf.unsorted_segment_sum(X, Y, k)
    counts = tf.unsorted_segment_sum(tf.ones_like(X), Y, k)
    return sums / counts

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    X = get_dataset(sess)
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations:  ❻
        i += 1
        Y = assign_cluster(X, centroids)
        centroids = sess.run(recompute_centroids(X, Y))
    print(centroids)
```

❶ Decide the number of clusters
❷ Declare the maximum number of iterations to run k-means
❸ Choose the initial guesses of cluster centroids
❹ Assign each data item to its nearest cluster
❺ Update the cluster centroids to their midpoint
❻ Iterate to find the best cluster locations

And that's it! If you know the number of clusters and the feature vector representation, you can use listing 5.4 to cluster anything! In the next section, we'll apply clustering to audio-snippets within an audio file.

5.4 Audio segmentation

In the last section, we clustered various audio files to automatically group them. This section is about using clustering algorithms within just one audio file. While the former is called clustering, the latter is referred to as segmentation. Segmentation is another word for

clustering, but we often say "segment" instead of "cluster" when dividing a single image or audio file into separate components. It's similar to how dividing a sentence into words is different from dividing up a word into letters. Though they both share a general idea of breaking bigger pieces into smaller components, words are very different from letters.

Let's say we have a long audio file, maybe of a podcast or talk show. Imagine writing a machine learning algorithm to identify which person is speaking in an audio interview between two people. The goal of segmenting an audio file is to associate which parts of the audio clip belong to the same category. In this case, there would be a category per each person, and the utterances made by each person should converge to their appropriate categories, as shown in figure 5.6.

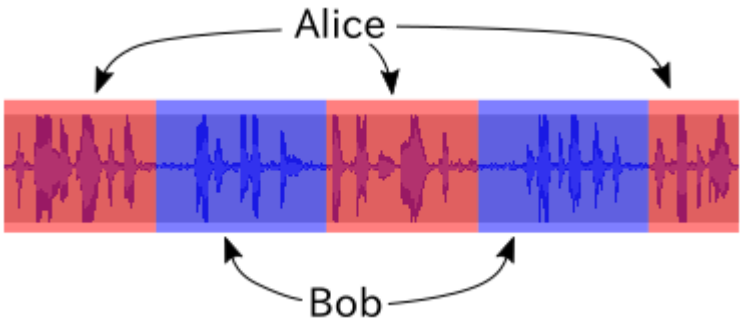


Figure 5.6 Audio segmentation is the process of automatically labelling segments.

Open a new source file and follow along with listing 5.5, which will get us started by organizing the audio data for segmentation. It splits up an audio file into multiple segments of size `segment_size`. A very long audio file should contain hundreds, if not thousands, of segments.

Listing 5.5 Organizing data for segmentation

```
import tensorflow as tf
import numpy as np
from bregman.suite import *

k = 4  ❶
segment_size = 50      ❷
max_iterations = 100  ❸

chromo = tf.placeholder(tf.float32)
max_freqs = tf.argmax(chromo, 0)

def get_chromogram(audio_file):
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)
    return F.X

def get_dataset(sess, audio_file):  ❹
    chromo_data = get_chromogram(audio_file)
    print('chromo_data', np.shape(chromo_data))
    chromo_length = np.shape(chromo_data)[1]
```

```
    xs = []
    for i in range(chromo_length / segment_size):
        chromo_segment = chromo_data[:, i*segment_size:(i+1)*segment_size]
        x = extract_feature_vector(sess, chromo_segment)
        if len(xs) == 0:
            xs = x
        else:
            xs = np.vstack((xs, x))
    return xs
```

1. Decide the number of clusters
2. The smaller the segment size, the better the results (but slower performance)
3. Decide when to stop the iterations
4. Obtain a dataset by extracting segments of the audio as separate data items

Now run the k-means clustering on this dataset to identify when segments are similar. The intention is that k-means will categorize similar sounding segments with the same label. If two people have significantly different sounding voices, then their sound-snippets will belong to different labels.

Listing 5.6 Segmenting an audio clip

```
with tf.Session() as sess:
    X = get_dataset(sess, 'sysk.wav')
    print(np.shape(X))
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations:  1
        i += 1
        Y = assign_cluster(X, centroids)
        centroids = sess.run(recompute_centroids(X, Y))
        if i % 50 == 0:
            print('iteration', i)
    segments = sess.run(Y)
    for i in range(len(segments)):               2
        seconds = (i * segment_size) / float(10)
        min, sec = divmod(seconds, 60)
        time_str = '{}m {}s'.format(min, sec)
        print(time_str, segments[i])
```

1. Run the k-means algorithm
2. Print the labels for each time interval

5.5 Clustering using a self-organizing map

A self-organizing map (SOM) is a model to represent data into a lower dimensional space. In doing so, it automatically shifts similar data items closer together. For example, suppose you're ordering pizza for a large gathering of people. You don't want to order the same type of pizza for every single person (because I happen to fancy pineapple with mushrooms and peppers for my toppings, though you may prefer anchovies with arugula and onions (I'm sorry)).

Each person's preference in his or her toppings can be represented as a three-dimensional vector. A SOM lets you to embed these three-dimensional vectors in two dimensions (as long as you define a distance metric between pizzas). Then, a visualization of the two-dimensional plot reveals good candidates for the number of clusters.

Although it may take longer to converge than the k-means algorithm, the SOM approach has no assumptions about the number of clusters. In the real-world, it's hard to select a value for the number of clusters. Consider a gathering of people as show in figure 5.7, in which the clusters change over time.

Figure 5.7 In the real world, we see groups of people in clusters all the time. Applying k-means requires knowing the number of clusters ahead of time. A more flexible tool is a self-organizing map, which has no preconceptions about the number of clusters.

The SOM merely re-interprets the data into a structure conducive to clustering. The algorithm works as follows. First, we design a grid of nodes, where each node holds a weight vector of the same dimension as a data item. The weights of each node are initialized to random numbers, typically from a standard normal distribution.

Next, we show data items to the network one by one. For each data item, the network identifies the node whose weight vector matches closest to it. This node is called the *best matching unit* (BMU).

After the network identifies the BMU, all neighbors of the BMU are updated so their weight vectors move closer to the BMU's value. The closer nodes are affected more strongly than nodes farther away. Moreover, the range of neighbors around a BMU shrinks over time at a rate determined usually by trial and error.

Listing 5.7 shows how to start implementing a SOM in TensorFlow. Follow along by opening a new source file.

Listing 5.7 Setting up the SOM algorithm

```
import tensorflow as tf
import numpy as np

class SOM:
    def __init__(self, width, height, dim):
        self.num_iters = 100
        self.width = width
        self.height = height
        self.dim = dim
        self.node_locs = self.get_locs()

        nodes = tf.Variable(tf.random_normal([width*height, dim]))     (1)
        self.nodes = nodes

        x = tf.placeholder(tf.float32, [dim])  (2)
        iter = tf.placeholder(tf.float32)      (2)

        self.x = x         (3)
        self.iter = iter   (3)

        bmu_loc = self.get_bmu_loc(x)          (4)

        self.propagate_nodes = self.get_propagation(bmu_loc, x, iter)  (5)
```

(1) Each node is a vector of dimension `dim`. For a 2D grid, there are `width * height` nodes
(2) These two ops are inputs at each iteration
(3) We'll need to access them from another method
(4) Find the node that matches closest to the input
(5) Update the values of the neighbors

Next, in listing 5.8, we define how to update neighboring weights given the current time interval and BMU location. As time goes by, the BMU's neighboring weights are less and less influenced to change. That way, over time the weights gradually settle.

Listing 5.8 Defining how to update the values of neighbors

```
    def get_propagation(self, bmu_loc, x, iter):
        num_nodes = self.width * self.height
        rate = 1.0 - tf.div(iter, self.num_iters)
        alpha = rate * 0.5
        sigma = rate * tf.to_float(tf.maximum(self.width, self.height)) / 2.
        expanded_bmu_loc = tf.expand_dims(tf.to_float(bmu_loc), 0)
        sqr_dists_from_bmu = tf.reduce_sum(
          tf.square(tf.sub(expanded_bmu_loc, self.node_locs)), 1)
        neigh_factor =
          tf.exp(-tf.div(sqr_dists_from_bmu, 2 * tf.square(sigma)))
        rate = tf.mul(alpha, neigh_factor)
        rate_factor = tf.pack([tf.tile(tf.slice(rate, [i], [1]), [self.dim]) for i in
       range(num_nodes)])
        nodes_diff = tf.mul(
          rate_factor,
          tf.sub(tf.pack([x for i in range(num_nodes)]), self.nodes))
        update_nodes = tf.add(self.nodes, nodes_diff)
        return tf.assign(self.nodes, update_nodes)
```

Listing 5.9 shows how to find the BMU location given an input data item. It searches through the grid of nodes to find the one with the closest match. This is very similar to the assignment step in k-means clustering, where each node in the grid is a potential cluster centroid.

Listing 5.9 Get the node location of the closest match

```
    def get_bmu_loc(self, x):
        expanded_x = tf.expand_dims(x, 0)
        sqr_diff = tf.square(tf.sub(expanded_x, self.nodes))
        dists = tf.reduce_sum(sqr_diff, 1)
        bmu_idx = tf.argmin(dists, 0)
        bmu_loc = tf.pack([tf.mod(bmu_idx, self.width), tf.div(bmu_idx, self.width)])
        return bmu_loc
```

In listing 5.10, we create a helper method to generate a list of *(x, y)* locations on all the nodes in the grid.

Listing 5.10 Generate a matrix of points

```
    def get_locs(self):
        locs = [[x, y]
                for y in range(self.height)
                for x in range(self.width)]
        return tf.to_float(locs)
```

Finally, let's define a method called `train` to run the algorithm, as shown in listing 5.11. First we must set up the session and run the `initialize_all_variables` op. Next, we loop `num_iters` number of times to update weights using the input data one by one. After the loop ends, we record the final node weights and their locations.

Listing 5.11 Run the SOM algorithm

```
    def train(self, data):
        with tf.Session() as sess:
            sess.run(tf.initialize_all_variables())
            for i in range(self.num_iters):
                for data_x in data:
                    sess.run(self.propagate_nodes, feed_dict={self.x: data_x, self.iter: i})
            centroid_grid = [[] for i in range(self.width)]
            self.nodes_val = list(sess.run(self.nodes))
            self.locs_val = list(sess.run(self.node_locs))
            for i, l in enumerate(self.locs_val):
                centroid_grid[int(l[0])].append(self.nodes_val[i])
            self.centroid_grid = centroid_grid
```

That's it! Now let's see it in action. Test the implementation by showing the SOM some input. In listing 5.12, our input is a list of 3-dimensional feature vectors. Training the SOM learns clusters within the data. We'll use a 4-by-4 grid, but it's best to try various values to cross-validate the best grid size. Figure 5.8 shows the output of running the code.

Listing 5.12 Test out and visualize results

```
from matplotlib import pyplot as plt
import numpy as np
from som import SOM

colors = np.array(
     [[0., 0., 1.],
      [0., 0., 0.95],
      [0., 0.05, 1.],
      [0., 1., 0.],
      [0., 0.95, 0.],
      [0., 1, 0.05],
      [1., 0., 0.],
      [1., 0.05, 0.],
      [1., 0., 0.05],
      [1., 1., 0.]])

som = SOM(4, 4, 3)  (1)
som.train(colors)

plt.imshow(som.centroid_grid)
plt.show()
```

(1) The grid size is 4x4, and the input dimension is 3

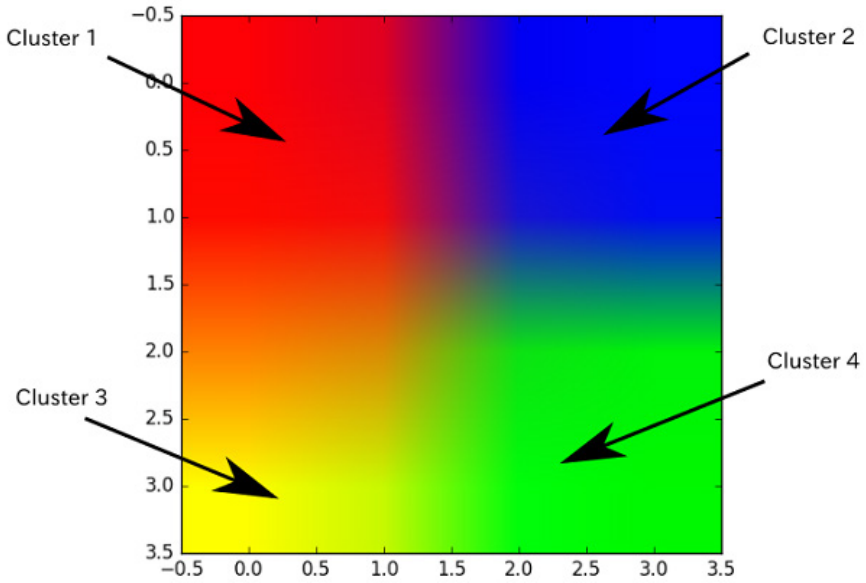


Figure 5.8 Here's a visualization of the SOM's output. It placed all 3-dimensional data points into a 2-dimensional grid. From it, you can pick the cluster centroids (automatically or manually) and achieve clustering in an intuitive lower-dimensional space.

The SOM embeds higher dimensional data into 2D to make clustering easy. This acts as a handy preprocessing step. You can manually go in and indicate the cluster centroids by observing the SOM's output, but it's also possible to automatically find good centroid candidates by observing the gradient of the weights.

5.6 Summary

We started the chapter with the intention of automatically organizing our music playlist by examining the audio data directly. The key tool was clustering, and we went through a couple types of algorithms. In summary,

- Clustering is an unsupervised machine learning algorithm to discover structure in data.
- K-means clustering is one of the easiest to implement and understand, and it also performs well in terms of speed and accuracy.
- If the number of clusters is not specified, we can use the self-organizing map algorithm to view the data in a simplified perspective.

While previous chapters discussed supervised learning, this chapter focused on unsupervised learning. In the next chapter, we'll see a machine learning algorithm that isn't really either of the two. It's a modeling framework that doesn't get as much attention by programmers nowadays, but is the essential tool for statisticians for unveiling hidden factors in data. Go ahead, flip the page and witness the awesomeness of Hidden Markov Models.

6

Reinforcement learning

This chapter covers

- Defining reinforcement learning
- Implementing reinforcement learning

Humans learn from past experiences (or, you know, at least they should). You didn't get so charming by accident. Years of positive compliments as well as negative criticism have all helped shape who you are today. This chapter is all about designing a machine learning system driven by criticisms and rewards.

Consider the following examples. You learn what makes people happy by interacting with friends, family, or even strangers, and you figure out how to ride a bike by trying out different muscle movements until it just clicks. When you perform actions, you're sometimes rewarded immediately. For example, finding a restaurant nearby might yield instant gratification. Other times, the reward doesn't appear right away, such as travelling a long distance to find an exceptional place to eat. Reinforcement learning is all about making the right actions given any state, such as in Figure 6.1 which shows a person making decisions to arrive at their destination.

Moreover, suppose on your drive from home to work you always choose the same route. But one day your curiosity takes over and you decide to try a different path in hopes for a shorter commute. This dilemma of trying out new routes or sticking to the best known route is an example of *exploration versus exploitation*.

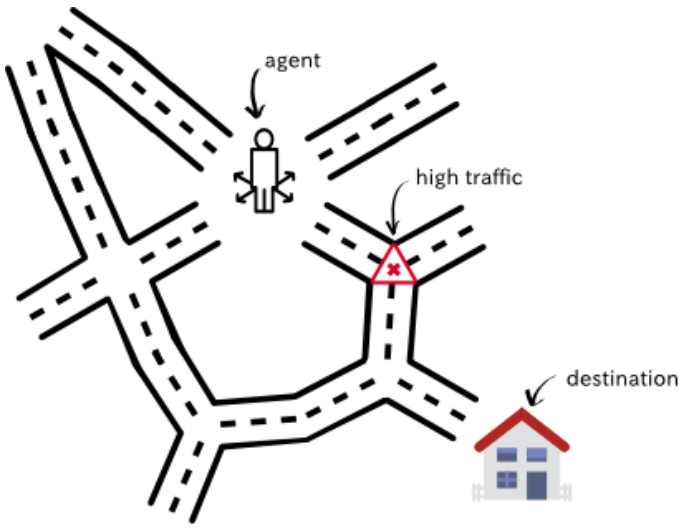


Figure 6.1 A person navigating his or her way to reach a destination in midst of traffic and unexpected situations is a problem set up for reinforcement learning.

ASIDE Why is the tradeoff between trying new things and sticking with old ones called "exploration versus exploitation"? Exploration makes sense, but you can think of exploitation as exploiting your knowledge of the status quo by sticking with what you know.

All these examples can be unified under a general formulation: performing an action in a scenario can yield a reward. A more technical term for scenario is *state*. And we call the collection of all possible states a *state-space*. Performing an action causes the state to change. But the question is, what series of actions yields the highest cumulative rewards?

6.1 Real-world examples

Reinforcement learning (RL) is used more often than you might expect. It's too easy to forget that it exists when you've learned supervised and unsupervised learning methods. But here are some examples to open your eyes to some successful uses of RL by Google:

- Game playing: In February 2015, Google developed a reinforcement learning system called DeepRL to automatically learn how to play arcade videogames from the Atari 2600 console. Unlike most RL solutions, this algorithm had a very high-dimensional input: it perceived the raw frame-by-frame images of the videogame. That way, the same algorithm could work with any videogame without much re-programming or re-configuring.
- More game playing: In January 2016, Google released a paper about an AI capable of winning the board game Go. The board game Go is known to be very unpredictable due to the enormous number of possible configurations (even more than Chess!), but this algorithm using RL was able to beat top human Go players.
- Robotics and control: In March 2016, Google demonstrated an efficient way for a robot to learn to grab an object by many examples. They collected over 800,000 grasp attempts using multiple robots and developed a model to grasp arbitrary objects. Impressively, the robots were capable of grasping an object from the help of a camera input alone.

6.2 Formal notions

Whereas supervised and unsupervised learning appear at opposite ends of the spectrum, reinforcement learning (RL) exists somewhere in the middle. It's not supervised learning, because the training data comes from the algorithm deciding between exploration and exploitation. And it's not unsupervised because the algorithm receives feedback from the environment. As long as you're in a situation where performing an action in a state produces a reward, you can use reinforcement learning to discover the best sequence of actions to take.

You may notice that reinforcement learning lingo involves anthropomorphizing the algorithm into taking "actions" in "situations" to "receive rewards." In fact, the algorithm is often referred to as an "agent" that "acts with" the environment. It shouldn't be a surprise

that much of reinforcement learning theory is applied in robotics. Figure 6.2 demonstrates the interplay between states, actions, and rewards.

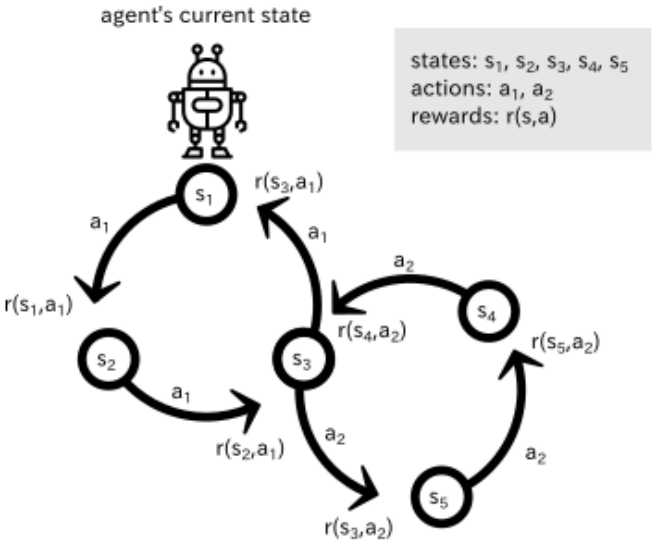


Figure 6.2 Actions are represented by arrows, and states are represented by circles. Performing an action on a state produces a reward. If you start at state s1, you can perform action a1 to obtain a reward r(s1, a1)

A robot performs actions to change between different states. But how does it decide which action to take? The next section introduces a new concept, called the *policy*, the answer this question.

Do humans use reinforcement learning?

Reinforcement learning seems like the best way to explain how to perform the next action based of the current situation. Perhaps humans behave the same way biologically. But let's not get ahead of ourselves and consider the following example.

Sometimes, humans act without thinking. If I'm thirsty, I might instinctively grab a cup of water to fulfil my thirst. I don't iterate through all possible joint motions in my head and choose the optimal one after thorough calculations. There seems to be a lot more going on, and a simple RL model might not fully explain human behavior.

6.3 Policy

Everyone cleans their room differently. Some people like to start by making their bed and spiraling out. I personally prefer cleaning my room clockwise so I don't miss a corner. Have you ever seen one of those robotic vacuum cleaners (such as Roomba)? Someone must have

programmed a strategy so that it can clean any room in which you place it. In reinforcement learning lingo, we call the strategy a *policy*.

One of the most common ways to solve reinforcement learning is by observing the long-term consequences of actions at each state. The short-term consequence is easy to calculate: that's just the reward. As you know, performing an action yields an immediate reward, but it's not always a good idea to greedily choose the action with the best reward all the time. That's a lesson in life too, because the most immediate best thing to do might not always be the most satisfying in the long run.

The best possible policy is often called the optimal policy, and it's often the holy grail of reinforcement learning. Learning the optimal policy, as shown in figure 6.3, tells you the optimal action given any state.

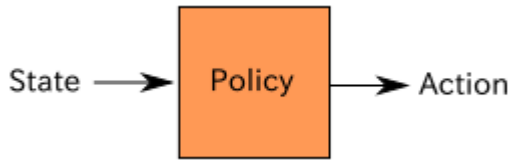


Figure 6.3 A policy suggests which action to take given a state.

As just mentioned, the way an agent decides which action to take is called its policy. We've so far described one type of policy where the agent always chooses the action with the greatest immediate reward, called the *greedy policy*. Another simple example of a policy is arbitrarily choosing an action, called the *random policy*. If you come up with a policy to solve a reinforcement learning problem, it's often a good idea to double-check that your learned policy performs better than a random policy.

6.4 Utility

The long-term reward is called a *utility*. It turns out, if we know the utility of performing an action at a state, then it's easy to solve reinforcement learning. For example, to decide which action to take, we simply select the action that produces the highest utility. The hard part, as you might have guessed, is uncovering these utility values.

The utility of performing an action *a* at a state *s* is written as a function Q(s, a), called the *utility function,* as shown in figure 6.4.

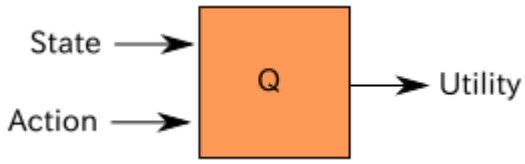


Figure 6.4 A utility function "Q" predicts the expected immediate reward plus rewards following an optimal policy given the state-action input.

EXERCISE 6.1 If you were given the utility function Q(s,a), how could you use it to derive a policy function?

An elegant way to calculate the utility of a particular state-action pair (s, a) is by recursively considering the utilities of future actions. The utility of your current action is influenced not by just the immediate reward but also the next best action, as shown in the formula below. In the formula, s' denotes the next state, and a' denotes the next action. The reward of taking action *a* in state *s* is denoted by *r(s, a):*

$$Q(s, a) = r(s, a) + \gamma \max Q(s', a')$$

Here, γ is a hyper-parameter that you get to choose, called the discount factor. If γ is 0, then the agent chooses the action that maximizes the immediate reward. Higher values of γ will make the agent put more importance in considering long-term consequences.

Looking ahead at future rewards is one type of hyper-parameter you can play with, but there's also another. In some applications of reinforcement learning, newly available information might be more important than historical records, or vice versa. For example, if a robot is expected to learn to solve tasks quickly but not necessarily optimally, we might want to set a faster learning rate. Or if a robot is allowed more time to explore and exploit, we might tune down the learning rate. Let's call the learning rate α, and change our utility function as follows (notice when α = 1, both equations are identical).

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r(s, a) + \gamma \max Q(s', a') - Q(s, a)\right)$$

Reinforcement learning can be solved if we know this Q(s, a) function. Conveniently for us, there's a machine learning strategy called *neural networks,* which are a way to approximate functions given enough training data. TensorFlow is the perfect tool to deal with neural networks because it comes with many essential algorithms to simplify neural network implementation.

6.5 Applying reinforcement learning

Application of reinforcement learning requires defining a way to retrieve rewards once an action is taken from a state. A stock market trader fits these requirements easily, because buying and selling a stock changes the state of the trader, and each action generates a reward (or loss).

The states in this situation are a vector containing information about the current budget, current number of stocks, and a recent history of stock prices (the last 200 stock prices). So each state is a 202-dimensional vector.

For simplicity, there are only three actions: buy, sell, and hold.

1. Buying a stock at the current stock price decreases the budget while incrementing the current stock count.
2. Selling a stock trades it in for money at the current share price.

3. Holding does neither, and performing that action simply waits a single time-period, and yields no reward.

Figure 6.5 demonstrates one possible policy given stock market data.

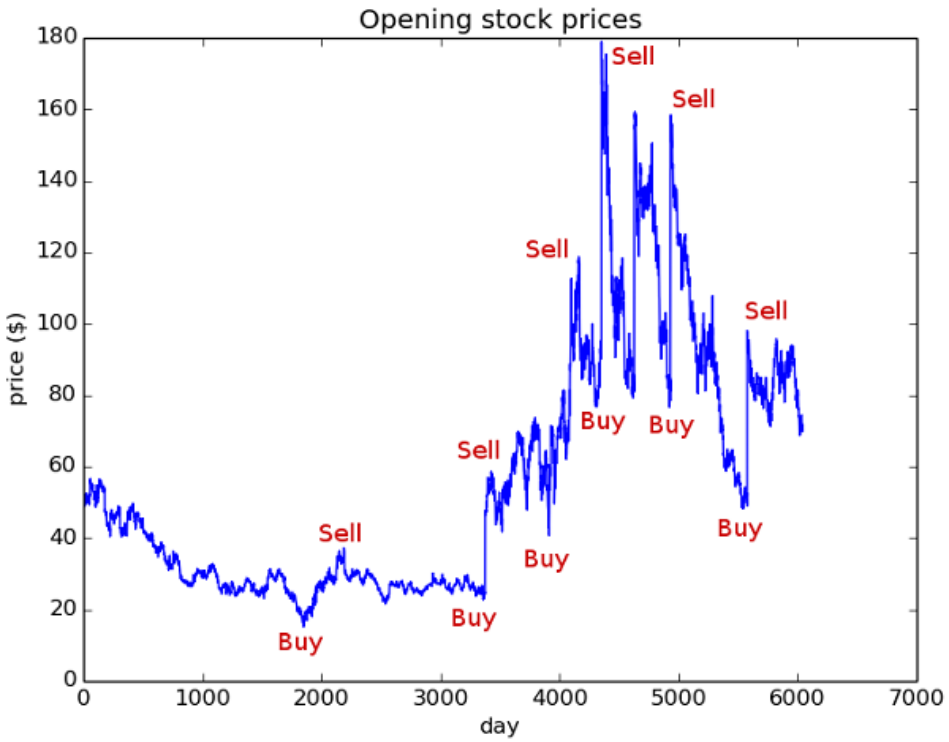


Figure 6.5 Ideally our algorithm should buy low and sell high. Doing so just once as shown in the figure yields a reward of around $120. But the real profit rolls in when you buy and sell more frequently. Ever heard the term high-frequency trading? It's about buying low and selling high as frequently as possible to maximize profits within a period of time.

The goal is to learn a policy that gains the maximum net-worth from trading in a stock market. Wouldn't that be cool? Let's do it!

6.6 Implementation

To gather stock prices, we will use the `yahoo_finance` library in Python. You can install it using pip, as shown below, or alternatively follow the official guide (https://pypi.python.org/pypi/yahoo-finance). The command to install it using `pip` is as follows.

```
$ pip install yahoo-finance
```

With that installed, let's import all the relevant libraries.

Listing 6.1 Import relevant libraries

```
from yahoo_finance import Share       (1)
from matplotlib import pyplot as plt  (2)
import numpy as np                    (3)
import tensorflow as tf               (3)
import random
```

(1) For obtaining stock price raw data
(2) For plotting stock prices
(3) For numeric manipulation and machine learning

Create a helper function to get stock prices using the `yahoo_finance` library. The library requires three pieces of information: share symbol, start date, and end date. When you pick each of the three values, you'll get a list of numbers representing the share prices in that period by day.

If you choose a start and end date too far apart, it'll take some time to fetch that data. It might be a good idea to save (that is, cache) the data to disk so you can load it locally next time. See listing 6.2 for how to use the library and cache the data.

Listing 6.2 Helper function to get prices

```
def get_prices(share_symbol, start_date, end_date,
               cache_filename='stock_prices.npy'):
    try:  (1)
        stock_prices = np.load(cache_filename)
    except IOError:
        share = Share(share_symbol)  (2)
        stock_hist = share.get_historical(start_date, end_date)
        stock_prices = [stock_price['Open'] for stock_price in stock_hist]  (3)
        np.save(cache_filename, stock_prices)  (4)

    return stock_prices
```

(1) Try to load the data from file if it has already been computed
(2) Retrieve stock prices from the library
(3) Extract only relevant info from the raw data
(4) Cache the result

Just for a sanity check, it's a good idea to visualize the stock price data. Create a plot and save it to disk.

Listing 6.3 Helper function to plot the stock prices

```
def plot_prices(prices):
    plt.title('Opening stock prices')
    plt.xlabel('day')
    plt.ylabel('price ($)')
```

```
    plt.plot(prices)
    plt.savefig('prices.png')
```

We can grab some data and visualize it using the following snippet of code in listing 6.4.

Listing 6.4 Get data and visualize it

```
if __name__ == '__main__':
  prices = get_prices('MSFT', '1992-07-22', '2016-07-22')
  plot_prices(prices)
```

Figure 6.4 shows the produced figure from running listing 6.6.

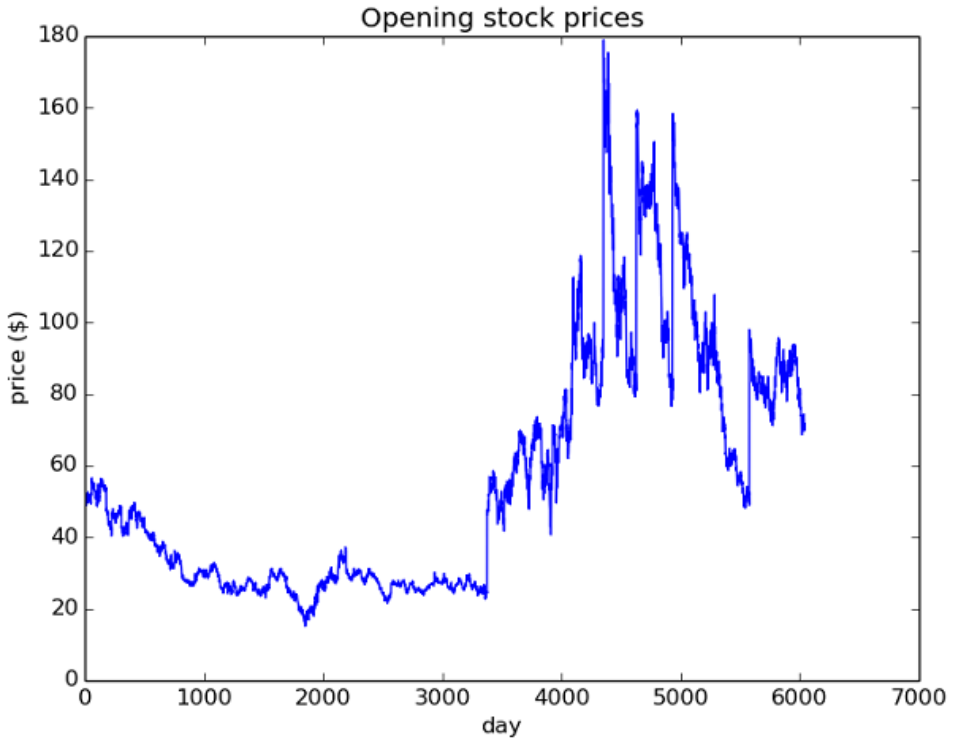


Figure 6.6 This chart summarizes the opening stock prices of Microsoft (MSFT) from 7/22/1992 to 7/22/2016. Wouldn't it have been nice to buy around day 3000 and sell around day 5000? Let's see if our code can learn to buy, sell, and hold to make the most money.

Most reinforcement learning algorithms follow similar implementation patterns. As a result, it's a good idea to create a class with the relevant methods to reference later, such as an abstract class or interface. See listing 6.5 for an example and figure 6.7 for an illustration. Basically,

reinforcement learning needs two operations well defined: (1) how to select an action, and (2) how to improve the utility Q-function.

Listing 6.5 Define a superclass for all decision policies

```
class DecisionPolicy:
    def select_action(self, current_state):  (1)
        pass

    def update_q(self, state, action, reward, next_state):  (2)
        pass
```

(1) Given a state, the decision policy will calculate the next action to take
(2) Improve the Q-function from a new experience of taking an action

$$\text{Infer}(s) \Rightarrow a$$

$$\text{Do}(s,\ a) \Rightarrow r,\ s'$$

$$\text{Learn}(s,\ r,\ a,\ s')$$

Figure 6.7 Most reinforcement learning algorithms boil down to just three main steps: infer, do, and learn. During the first step, the algorithm selects the best action (a) given a state (s) using the knowledge it has so far. Next, it does the action to find out the reward (r) as well as the next state (s'). Then it improves its understanding of the world using the newly acquired knowledge (s, r, a, s').

Next, let's inherit from this superclass to implement a random decision policy. We only need to define the `select_action` method, which will randomly pick an action without even looking at the state. See listing 6.6 for how to implement it.

Listing 6.6 Implement a random decision policy

```
class RandomDecisionPolicy(DecisionPolicy):  (1)
    def __init__(self, actions):
        self.actions = actions

    def select_action(self, current_state):  (2)
        action = self.actions[random.randint(0, len(self.actions) - 1)]
        return action
```

(1) Inherit from DecisionPolicy to implement its functions
(2) Randomly choose the next action

In listing 6.7, we assume a policy is given to us (such as the one from listing 6.6), and run it on the real world stock-price data. This function takes care of exploration and exploitation at each interval of time. Figure 6.8 illustrates the algorithm from listing 6.7.

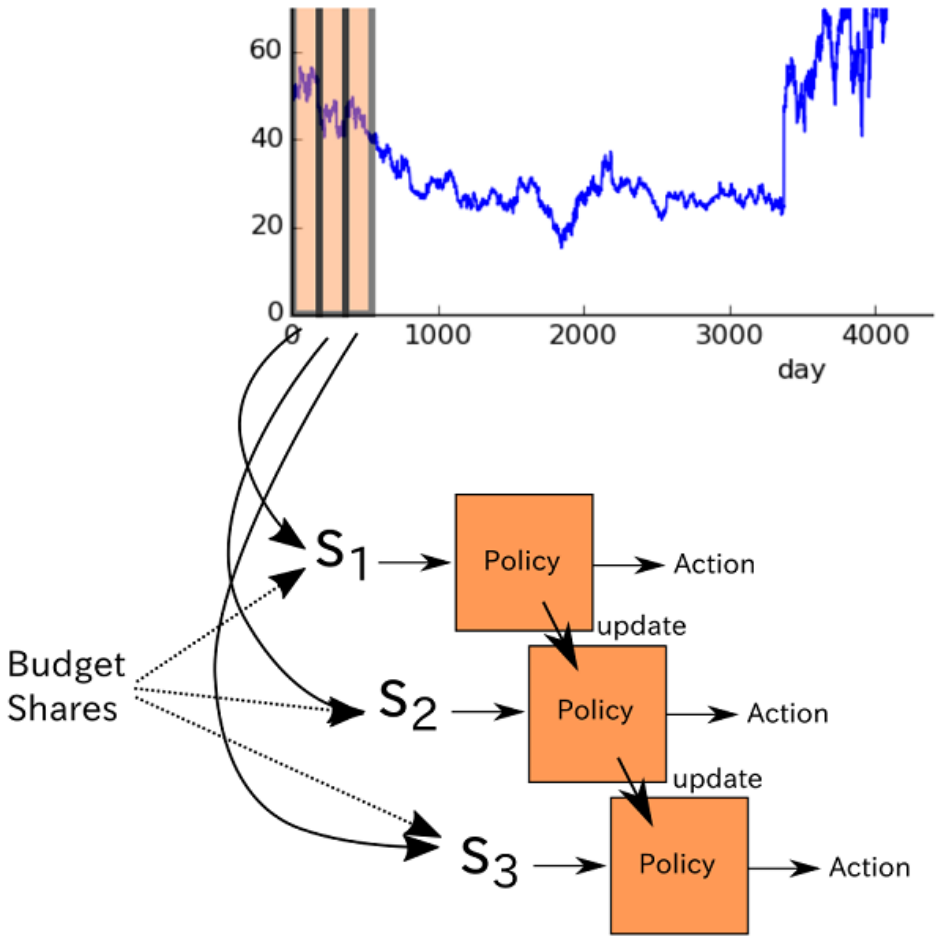


Figure 6.8 A rolling window of some size iterates through the stock prices as shown by the chart segmented to form states s1, s2, and s3. The policy suggests an action to take, which we may either choose to exploit or otherwise randomly explore another action. As we get rewards for performing an action, we can update the policy function over time.

Listing 6.7 Use a given policy to make decisions and return the performance

```
def run_simulation(policy, initial_budget, initial_num_stocks, prices, hist,
                   debug=False):
    budget = initial_budget  (1)
    num_stocks = initial_num_stocks  (1)
    share_value = 0          (1)
    transitions = list()
    for i in range(len(prices) - hist - 1):
        if i % 100 == 0:
            print('progress {:.2f}%'.format(float(100*i) / (len(prices) - hist - 1)))
        current_state = np.asmatrix(np.hstack((prices[i:i+hist], budget, num_stocks)))  (2)
        current_portfolio = budget + num_stocks * share_value  (3)
        action = policy.select_action(current_state, i)        (4)
        share_value = float(prices[i + hist + 1])
        if action == 'Buy' and budget >= share_value:          (5)
            budget -= share_value
            num_stocks += 1
        elif action == 'Sell' and num_stocks > 0:              (5)
            budget += share_value
            num_stocks -= 1
        else:  (5)
            action = 'Hold'
        new_portfolio = budget + num_stocks * share_value      (6)
        reward = new_portfolio - current_portfolio             (7)
        next_state = np.asmatrix(np.hstack((prices[i+1:i+hist+1], budget, num_stocks)))
        transitions.append((current_state, action, reward, next_state))
        policy.update_q(current_state, action, reward, next_state)  (8)

    portfolio = budget + num_stocks * share_value  (9)
    if debug:
        print('${}\t{} shares'.format(budget, num_stocks))
    return portfolio
```

(1) Initialize values that depend on computing the net worth of a portfolio
(2) The state is a `hist+2` dimensional vector
(3) Calculate the portfolio value
(4) Select an action from the current policy
(5) Update portfolio values based on action
(6) Compute new portfolio value after taking action
(7) Compute the reward from taking an action at a state
(8) Update the policy after experiencing a new action
(9) Compute final portfolio worth

To obtain a more robust measurement of success, let's run the simulation a couple times and average the results. Doing so may take a while to complete (perhaps 5 minutes), but your results will be more reliable.

Listing 6.8 Run multiple simulations to calculate an average performance

```
def run_simulations(policy, budget, num_stocks, prices, hist):
    num_tries = 10  (1)
    final_portfolios = list()  (2)
    for i in range(num_tries):
```

```
        final_portfolio = run_simulation(policy, budget, num_stocks, prices, hist)  ❸
        final_portfolios.append(final_portfolio)
    avg, std = np.mean(final_portfolios), np.std(final_portfolios)  ❹
    return avg, std
```

❶ Decide number of times to re-run the simulations
❷ Store portfolio worth of each run in this array
❸ Run this simulation
❹ Average the values from all the runs

In main, define the decision policy and try running simulations to see how it performs.

Listing 6.9 Append the following lines to main

```
if __name__ == '__main__':
    prices = get_prices('MSFT', '1992-07-22', '2016-07-22')
    plot_prices(prices)
    actions = ['Buy', 'Sell', 'Hold']       ❶
    hist = 200
    policy = RandomDecisionPolicy(actions)  ❷
    budget = 1000.0    ❸
    num_stocks = 0     ❹
    avg, std = run_simulations(policy, budget, num_stocks, prices, hist)  ❺
    print(avg, std)
```

❶ Define the list of actions the agent can take
❷ Initial a random decision policy
❸ Set the initial amount of money available to use
❹ Set the number of stocks already owned
❺ Run simulations multiple times to compute expected value of final net worth

Now that we have a baseline to compare our results, let's implement our neural network approach to learn the Q-function. The decision policy is often called the Q-learning decision policy. The following listing 6.10 introduces a new hyper-parameter "epsilon" to keep the solution from getting "stuck" when applying the same action over and over. The lesser its value, the more often it will randomly explore new actions. The Q-function is defined by the function visualized in figure 6.8.

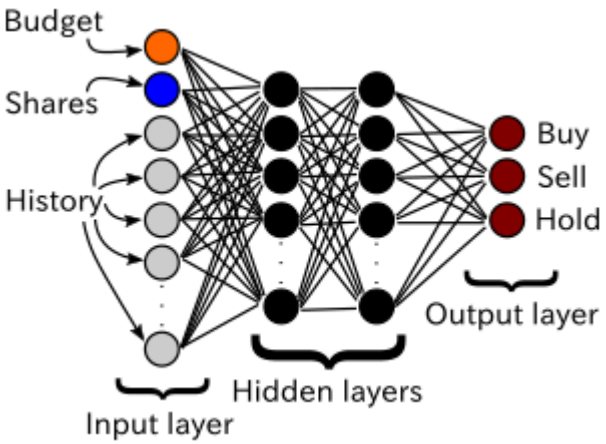


Figure 6.9 The input is the state space vector, with three outputs, one for each output's Q-value.

Listing 6.10 Implement a more intelligent decision policy

```
class QLearningDecisionPolicy(DecisionPolicy):
    def __init__(self, actions, input_dim):
        self.epsilon = 0.9  (1)
        self.gamma = 0.01   (1)
        self.actions = actions
        output_dim = len(actions)
        h1_dim = 200        (2)

        self.x = tf.placeholder(tf.float32, [None, input_dim])    (3)
        self.y = tf.placeholder(tf.float32, [output_dim])         (3)
        W1 = tf.Variable(tf.random_normal([input_dim, h1_dim]))   (4)
        b1 = tf.Variable(tf.constant(0.1, shape=[h1_dim]))        (4)
        h1 = tf.nn.relu(tf.matmul(self.x, W1) + b1)   (4)
        W2 = tf.Variable(tf.random_normal([h1_dim, output_dim]))  (4)
        b2 = tf.Variable(tf.constant(0.1, shape=[output_dim]))    (4)
        self.q = tf.nn.relu(tf.matmul(h1, W2) + b2)   (5)

        loss = tf.square(self.y - self.q)  (6)
        self.train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)  (7)
        self.sess = tf.Session()           (8)
        self.sess.run(tf.initialize_all_variables())  (8)

    def select_action(self, current_state, step):
        threshold = min(self.epsilon, step / 1000.)
        if random.random() < threshold:    (9)
            # Exploit best option with probability epsilon
            action_q_vals = self.sess.run(self.q, feed_dict={self.x: current_state})
            action_idx = np.argmax(action_q_vals)
            action = self.actions[action_idx]
        else:  (10)
            # Explore random option with probability 1 - epsilon
            action = self.actions[random.randint(0, len(self.actions) - 1)]
        return action
```

```
def update_q(self, state, action, reward, next_state):  ⓫
    action_q_vals = self.sess.run(self.q, feed_dict={self.x: state})
    next_action_q_vals = self.sess.run(self.q, feed_dict={self.x: next_state})
    next_action_idx = np.argmax(next_action_q_vals)
    action_q_vals[0, next_action_idx] = reward + self.gamma * next_action_q_vals[0,
   next_action_idx]
    action_q_vals = np.squeeze(np.asarray(action_q_vals))
    self.sess.run(self.train_op, feed_dict={self.x: state, self.y: action_q_vals})
```

❶ Set the hyper-parameters from the Q-function
❷ Set the number of hidden nodes in the neural networks
❸ Define the input and output tensors
❹ Design the neural network architecture
❺ Define the op to compute the utility
❻ Set the loss as the square error
❼ Use an optimizer to update model parameters to minimize the loss
❽ Set up the session and initialize variables
❾ Exploit best option with probability epsilon
❿ Explore random option with probability 1-epsilon
⓫ Update the Q-function by updating its model parameters

6.7 Summary

Now that we've applied reinforcement learning to the stock market, it's time for you to drop out of school or quit your job and start gaming the system. Turns out this is your payoff, dear reader, for making it this far into the book! Just kidding, the actual stock market is a much more complicated beast, but the techniques used in this chapter generalize to many situations.

- Reinforcement learning is the natural tool when a problem can be framed by states that change due to actions that can be taken by an agent to discover rewards.
- There are three primary steps in implementing the algorithm: infer the best action from the current state, do the action, and learn from the results.
- Q-learning is an approach to solving reinforcement learning where you develop an algorithm to approximate the utility function (Q-function). Once a good enough approximation is found, you can start inferring best actions to take from each state.

7

Hidden Markov models

This chapter covers:

- Defining interpretive models
- Using Markov chains to model data
- Inferring hidden state using a Hidden Markov Model

If a rocket blows up, someone's probably getting fired, so rocket scientists and engineers must be able to make confident decisions about all components and configurations. They do so by physical simulations and mathematical deduction from first principles. You, too, have solved science problems with pure logical thinking. Consider Boyle's law: pressure and volume of a gas are inversely related under a fixed temperature. You can make insightful inferences from these simple laws that have been discovered about the world. Recently, machine learning has started to play the role of an important side-kick to deductive reasoning.

"Rocket science" and "machine learning" aren't phrases that usually appear together. But nowadays, modeling real-world sensor readings using intelligent data-driven algorithms is more approachable in the aerospace industry. Also, the use of machine learning techniques is flourishing in the healthcare and automotive industries. But why?

Part of the reason for this influx can be attributed to better understanding of *interpretable* models, which are machine learning models where the learned parameters have clear interpretations. If a rocket blows up, for example, an interpretable model might help trace the root cause.

> **EXERCISE 7.1** What makes a model interpretable may be slightly subjective. What is your criteria for an interpretable model?

This chapter is about exposing the hidden explanations behind observations. Consider a puppet-master pulling strings to make a puppet appear alive. Analyzing only the motions of the puppet might lead to over-complicated conclusions about how it's possible for an inanimate object to move. Once you notice the attached strings, you'll realize that a puppet-master is the best explanation for the life-like motions.

On that note, this chapter introduces Hidden Markov Models (HMM), which reveal intuitive properties about the problem under study. The HMM is the "puppet-master," which explains the observations. We model observations using Markov chains, which will be described in section 7.2.

Before going into details about Markov chains and HMMs, let's consider some alternative models. Follow along to section 7.1 to witness how some models may not be interpretable.

7.1 Example of a not-so-interpretable model

Here's a classic example of a block-box machine learning algorithm that is difficult to interpret. In an image classification task, the goal is to assign a label to each image. More simply, image classification is often posed as a multiple choice question: "which one of the listed categories best describes the image." Machine learning practitioners have made tremendous advancements in solving this problem. Today's best image classifiers match human-level performance on certain datasets.

You'll learn how to solve this problem in the later chapters on convolutional neural networks (CNN), which are a class of machine learning models that end up learning a lot of parameters. But that's also the problem with CNNs: what do each of the thousands, if not millions, of parameters mean? It's difficult to ask an image classifier why it made the decision that it did. All we have available are the learned parameters, which do not easily explain the reasoning behind the classification.

Machine learning sometimes gets the notoriety of being a black-box tool that solves a specific problem without revealing insight on how it arrives at its conclusion. The purpose of this chapter is to unveil an area of machine learning with an interpretable model. Specifically we'll learn about the HMM and use TensorFlow to implement it.

7.2 Markov Model

Andrey Markov was a Russian mathematician who studied how systems change over time in the presence of randomness. Imagine gas particles bouncing around in the air. Tracking the position of each particle by Newtonian physics can get way too complicated, so introducing randomness helps simplify the physical model a little.

Markov realized that what helps simplify a random system even further is if you consider only a neighborhood around the gas particle to model it. For example, maybe a gas particle in Europe has barely any effect on a particle in the United States. So why not just ignore it? The mathematics simplifies when you only look at a nearby neighborhood instead of the entire system. This notion is now referred to as the *Markov property*.

Consider modeling the weather. A meteorologist evaluates various conditions involving thermometers, barometers, and anemometers to help predict the weather. They draw upon brilliant insight and years of experience to do their job.

Let's see how we can use the Markov property to help us get started with a simple model. First, we identify the possible states that we care to study. Figure 7.1 shows three weather conditions as nodes in a graph: cloudy, rainy, and sunny.

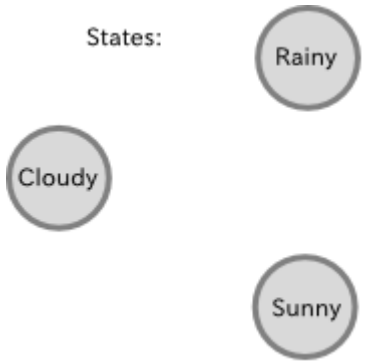


Figure 7.1 Weather conditions represented as nodes in a graph.

Now that we have our states, we want to also define how one state transforms into another. It's difficult to model weather as a deterministic system. It's not obvious to say that if it were sunny today, it will certainly be sunny again tomorrow. Instead, we can introduce randomness and say if it were sunny today, there's a 90% chance it'll be sunny again tomorrow, and a 10% chance it will be cloudy. The Markov property comes in play when we only use today's weather condition to predict tomorrow's (instead of using all previous history).

EXERCISE 7.2 A robot that decides which action to perform based on only its current state is said to follow the Markov property. What are the advantages and disadvantages of such a decision making process?

Figure 7.2 demonstrates the transitions as directed edges drawn between nodes. Each edge has a weight representing the probability. The lack of an edge between two nodes is an elegant way of showing that the probability of that transformation is near zero. The transition probabilities can be learned from historical data, but for now, let's just assume they're given to us.

If you have three states, you can represent the transitions as a 3x3 matrix. Each element of the matrix (at row i and column j) corresponds to the probability associated with the edge from node i to node j. In general, if you have N states, then the *transition matrix* will be NxN in size.

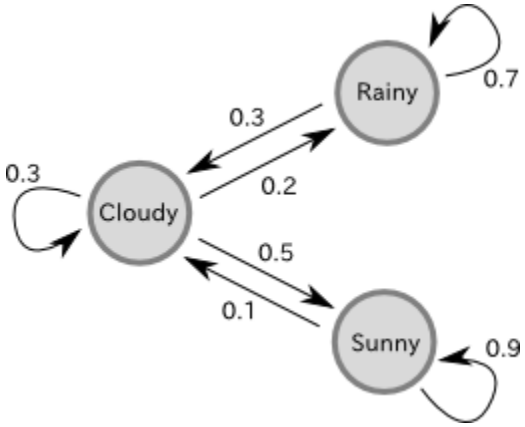


Figure 7.2 Transition probabilities between weather conditions are represented as directed edges.

We call this system a *Markov model*. Over time, a state changes using the transition probabilities previously defined (as shown in Figure 7.2). Figure 7.3 is another way to visualize how the states change given the transition probabilities. It's often called a *trellis diagram*, and turns out to be an essential tool in later implementing the TensorFlow algorithms.

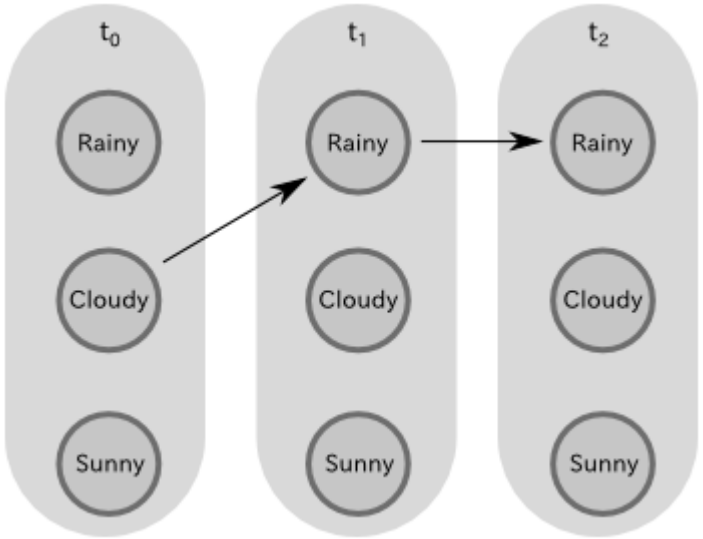


Figure 7.3 A trellis representation of the Markov system changing states over time.

You've seen in the previous chapters how TensorFlow code builds a graph to represent computation. It might be tempting to treat each node in a Markov model as a node in

TensorFlow. However, even though figures 7.2 and 7.3 nicely visualize state transitions, there's a more efficient way they will actually be implemented in code, as shown in figure 7.4.

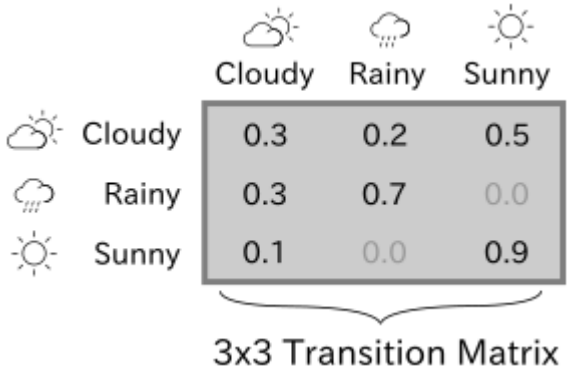


Figure 7.4 A transition matrix conveys the probabilities of a state from the left (rows) transitioning to a state to the top (columns).

Remember, nodes in a TensorFlow graph are Tensors, so we can represent a transition matrix (let's call it *T*) as simply a node in TensorFlow. Then we can apply mathematical operations on the TensorFlow node to achieve interesting results.

For example, suppose you prefer sunny days over rainy ones, so you have a score associated to each day. You represent your scores for each state in a 3x1 matrix called *s*. Then multiplying the two matrices in TensorFlow `tf.matmul(T*s)` gives the expected preference of transitioning from each state.

Representing a scenario in a Markov model allows us to great simplify how we view the world. However, it's often difficult to measure the state of the world directly. Often times, we have to use evidence from multiple observations to figure out the hidden meaning. And that's what the next section aims to solve!

7.3 Hidden Markov Model

The Markov model defined in the previous section is convenient when all the states are observable, but that's not always the case. Consider having access to only temperate readings of a town. How, then, could you infer the weather given only derived data?

Rainy weather most likely causes a lower temperature reading, whereas a sunny day most likely causes a higher temperature reading. With temperature knowledge and transition probabilities alone, you can still make intelligent inferences on the most likely weather. Problems like this are very common in the real world. A state might leave traces of hints behind, and those hints are all you have available to you.

Models like these are called *Hidden Markov Models* (HMM), because the true states of the world (such as whether it's raining or sunny) are not directly observable. These hidden states follow a Markov model, and each state emits a measurable observation with some likelihood.

For example, the hidden state of "Sunny" might emit high temperature readings, but occasionally also low readings for one reason or another.

In a HMM, we have to define the emission probability, which is usually represented as a matrix, called the *emission matrix*. The number of rows of the matrix is the number of states (Sunny, Cloudy, Rainy), and the number of columns is the number of different types of observations (Hot, Mild, Cold). Each element of the matrix is the probability associated with the emission.

The canonical way of visualizing a HMM is by appending the trellis with observations as shown in figure 7.5.

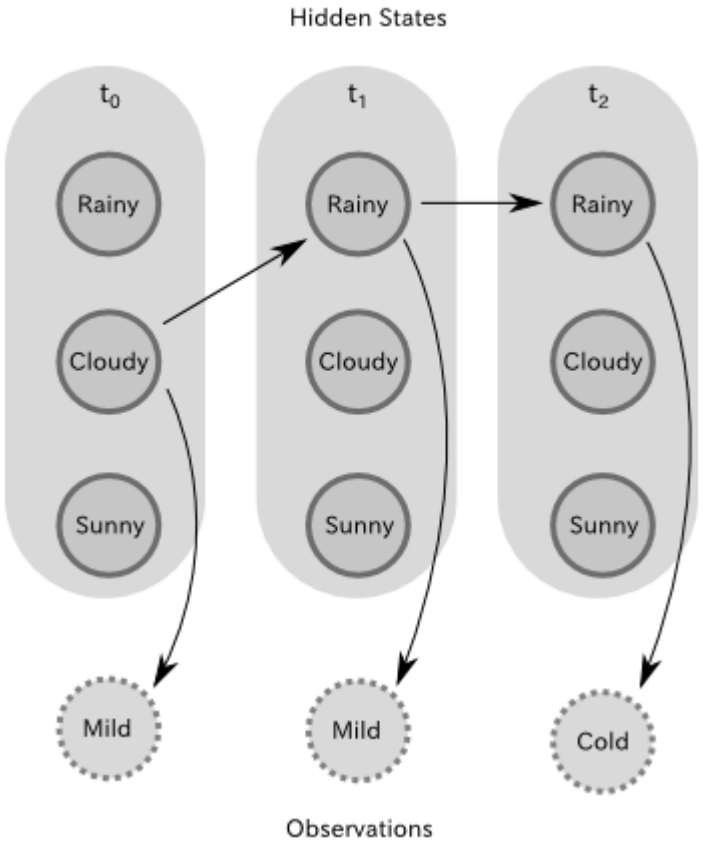


Figure 7.5 A Hidden Markov Model trellis showing how weather conditions might produce temperature readings.

So that's almost it. The HMM is a description about transition probabilities, emission probabilities, and one more thing: *initial probabilities*. The initial probability is the probability of each state happening with no prior knowledge. If we're modeling the weather in Los

Angeles, then perhaps the initial probability of "Sunny" would be much greater. Or, let's say you're modelling the weather in Seattle; well you know, you can set the initial probability of "Rainy" to something higher.

A HMM lets us understand a sequence of observations. In this weather modelling scenario, one question we may ask is what's the probability of observing a certain sequence of temperature readings? We'll answer this question using what is known as the *Forward algorithm*.

7.4 Forward algorithm

The forward algorithm computes the probability of an observation. There's many permutations that may cause a particular observation, so enumerating all possibilities the naïve way will take an exponentially long time to compute.

Instead, we can solve the problem using dynamic programming, which is essentially the strategy of breaking up a complex problem into simple little ones and using a look-up table to cache the results. In our code we'll save the lookup table as a NumPy array and feed it to a TensorFlow op to keep updating it.

As shown in listing 7.1, create a HMM class to capture the Hidden Markov Model parameters, which include the initial probability vector, transition probability matrix, and emission probability matrix.

Listing 7.1 Define the HMM class

```
import numpy as np          (1)
import tensorflow as tf     (1)


class HMM(object):
    def __init__(self, initial_prob, trans_prob, obs_prob):
        self.N = np.size(initial_prob)              (2)
        self.initial_prob = initial_prob            (2)
        self.trans_prob = trans_prob                (2)
        self.emission = tf.constant(obs_prob)       (2)

        assert self.initial_prob.shape == (self.N, 1)       (3)
        assert self.trans_prob.shape == (self.N, self.N)    (3)
        assert obs_prob.shape[0] == self.N                  (3)

        self.obs_idx = tf.placeholder(tf.int32)     (4)
        self.fwd = tf.placeholder(tf.float64)       (4)
```

(1) Import the required libraries
(2) store the parameters as method variables
(3) double-check the shapes of all the matricies makes sense
(4) define the placeholders used for the forward algorithm

Next, we'll define a quick helper function in listing 7.2 to access a row from the emission matrix.

Listing 7.2 Create a helper function to access emission probability of an observation

```
    def get_emission(self, obs_idx):
        slice_location = [0, obs_idx]  ❶
        num_rows = tf.shape(self.emission)[0]
        slice_shape = [num_rows, 1]    ❷
        return tf.slice(self.emission, slice_location, slice_shape)  ❸
```

❶ The location of where to slice the emission matrix
❷ The shape of the slice
❸ Perform the slicing operator

We'll need to define two TensorFlow ops. The first one (in listing 7.3) will be run only once to initialize the forward algorithm's cache.

Listing 7.3 Initializing the cache

```
    def forward_init_op(self):
        obs_prob = self.get_emission(self.obs_idx)
        fwd = tf.mul(self.initial_prob, obs_prob)
        return fwd
```

And the next op will update the cache at each observation, as shown in listing 7.4.

Listing 7.4 Updating the cache

```
    def forward_op(self):
        transitions = tf.matmul(self.fwd, tf.transpose(self.get_emission(self.obs_idx)))
        weighted_transitions = transitions * self.trans_prob
        fwd = tf.reduce_sum(weighted_transitions, 0)
        return tf.reshape(fwd, tf.shape(self.fwd))
```

Outside of the HMM class, let's define a function to run the forward algorithm, as shown in listing 7.5.

Listing 7.5 Defining the forward algorithm given a HMM

```
def forward_algorithm(sess, hmm, observations):
    fwd = sess.run(hmm.forward_init_op(), feed_dict={hmm.obs_idx: observations[0]})
    for t in range(1, len(observations)):
        fwd = sess.run(hmm.forward_op(), feed_dict={hmm.obs_idx: observations[t], hmm.fwd:
       fwd})
    prob = sess.run(tf.reduce_sum(fwd))
    return prob
```

In the main function, let's set up the HMM class by feeding it the initial probability vector, transition probability matrix, and emission probability matrix. Then we'll call the forward algorithm that we just defined above. See listing 7.6.

Listing 7.6 Define the HMM and call the forward algorithm

```
if __name__ == '__main__':
    initial_prob = np.array([[0.6], [0.4]])
    trans_prob = np.array([[0.7, 0.3], [0.4, 0.6]])
    obs_prob = np.array([[0.5, 0.4, 0.1], [0.1, 0.3, 0.6]])

    hmm = HMM(initial_prob=initial_prob, trans_prob=trans_prob, obs_prob=obs_prob)

    observations = [0, 1, 1, 2, 1]
    with tf.Session() as sess:
        prob = forward_algorithm(sess, hmm, observations)
        print('Probability of observing {} is {}'.format(observations, prob))
```

By running listing 7.6, the algorithm will output the following:

```
Probability of observing [0, 1, 1, 2, 1] is 0.004642
```

7.5 Viterbi decode

The Viterbi decoding algorithm finds the most likely sequence of hidden states given a sequence of observations. It'll require a caching scheme similar to the forward algorithm. We'll name the cache `viterbi`. In the HMM constructor, append the following line shown in listing 7.7.

Listing 7.7 Add the Viterbi cache as a member variable

```
def __init__(self, initial_prob, trans_prob, obs_prob):
  ...
  ...
  ...
  self.viterbi = tf.placeholder(tf.float64)
```

In listing 7.8, let's define a TensorFlow op to update the `viterbi` cache. This will be a method in the HMM class.

Listing 7.8 Define an op to update the forward cache

```
def decode_op(self):
        transitions = tf.matmul(self.viterbi, tf.transpose(self.get_emission(self.obs)))
        weighted_transitions = transitions * self.trans_prob
        viterbi = tf.reduce_max(weighted_transitions, 0)
        return tf.reshape(viterbi, tf.shape(self.viterbi))
```

We'll also need an op to update the back pointers.

Listing 7.9 Define an op to update the back pointers

```
def backpt_op(self):
    back_transitions = tf.matmul(self.viterbi, np.ones((1, self.N)))
    weighted_back_transitions = back_transitions * self.trans_prob
```

```
    return tf.argmax(weighted_back_transitions, 0)
```

Lastly, in listing 7.10, define the Viterbi decoding function outside of the HMM.

Listing 7.10

```
def viterbi_decode(sess, hmm, observations):
    viterbi = sess.run(hmm.forward_init_op(), feed_dict={hmm.obs: observations[0]})
    backpts = np.ones((hmm.N, len(observations)), 'int32') * -1
    for t in range(1, len(observations)):
        viterbi, backpt = sess.run([hmm.decode_op(), hmm.backpt_op()],
                                   feed_dict={hmm.obs: observations[t],
                                              hmm.viterbi: viterbi})
        backpts[:, t] = backpt
    tokens = [viterbi[:, -1].argmax()]
    for i in range(len(observations) - 1, 0, -1):
        tokens.append(backpts[tokens[-1], i])
    return tokens[::-1]
```

We can run the code in listing 7.11 in the main function to evaluate the Viterbi decoding of an observation.

Listing 7.11 Run the Viterbi decode

```
seq = viterbi_decode(sess, hmm, observations)
print('Most likely hidden states are {}'.format(seq))
```

7.6 Uses of Hidden Markov Models

Now that we've implemented the forward-pass and Viterbi algorithms, let's take a look at some interesting uses for our new found power.

1. Modelling a video:

   Imagine being able to recognize a person based solely (no pun intended) on how they walk. Identifying people based on their gait is a pretty cool idea, but first we need a model to recognize the gate. Consider a HMM where the sequence of hidden states for a gate are (1) rest position, (2) right foot forward, (3) rest position, (4) left foot forward, and finally (5) rest position. The observed states are silhouettes of a person walking/jogging/running taken from a video clip (here's a dataset of such examples http://figment.csee.usf.edu/GaitBaseline/).

2. Modelling DNA:

   DNA is a sequence of nucleotides, and we're gradually learning more about its structure. One clever way to understand a long DNA string is by modeling the regions if we know some probability about the order they appear. Just like how cloudy days are common after a rainy day, maybe a certain region on the DNA sequence (start codon) is more common before another region (stop codon).

3. Modelling an image:

In handwriting recognition, we aim to retrieve the plaintext from an image of handwritten words. One approach is resolve characters one at a time, and then concatenate the results. We can use the insight that are written in a sequence to build a HMM. Knowing the previous character could probably help us rule out possibilities of the next character. The hidden states are the plaintext, and the observations are cropped images containing individual characters.

7.7 Summary

You now have what it takes to design your own experiments using Hidden Markov Models! It's a powerful tool, and I urge you to try it on your own data. Pre-define some transitions and emissions, and see if you can recover hidden states. Hopefully this chapter can help get you started.

The study of HMMs is ever-expanding, with many new ideas and modifications made all the time. Amongst all this rapid development, here are some key take-aways.

- A complicated entangled system can be simplified using a Markov model.
- The Hidden Markov Model ends up being more useful in real-world applications because most observations are measurements of hidden states.
- The forward-pass and Viterbi algorithm are among the most common algorithms used on HMMs

8

A peek into autoencoders

This chapter covers

- Introduction to neural networks
- Designing autoencoders
- Representing images

Have you ever identified a song from a person just humming a melody? It might be easy for you, but I'm comically tone-deaf when it comes to music. Humming, by itself, is an approximation of its corresponding song. An even better approximation could be singing. Include some instrumentals, and sometimes a cover of a song sounds indistinguishable from the original.

Instead of songs, in this chapter, we will approximate functions. Functions are a very general notion of relations between inputs and outputs. In machine learning, we typically want to find the function that relates inputs to outputs. Finding the best possible function is difficult, but approximating the function is much easier.

Conveniently, artificial neural networks are a model in machine learning that can approximate any function. Given training data, we want to build a neural network model that best approximates the implicit function that might have generated the data.

After introducing neural networks in section 8.1, we'll learn how to use them to encode data into a smaller representation in section 8.2, using a network structure called an autoencoder.

8.1 Neural Networks

If you've ever heard about neural networks, you've probably seen diagrams of nodes and edges connected in a complicated mesh. The motivation for that visualization is mostly inspired by biology, specifically neurons in the brain. Turns out, it's also a convenient way to visualize functions, like $f(x) = w * x + b$ shown in figure 8.1.

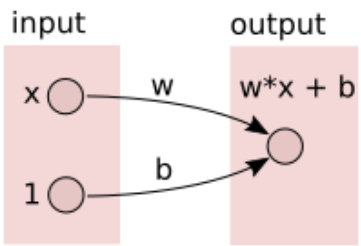


Figure 8.1 A graphical representation of a linear equation f(x) = w*x + b.

As a reminder, a linear model is set of linear functions; for example, $f(x) = w * x + b$, where (w, b) is the vector of parameters. The learning algorithm drifts around the values of w and b until it finds a combination that best matches the data. Once the algorithm successfully converges, it'll find the best possible linear function to describe the data.

Linear is a good first start, but the real world isn't always that pretty. And thus, we dive into the type of machine learning responsible for TensorFlow's inception; this chapter will be our first introduction to a type of model called an *artificial neural network*, which can approximate arbitrary functions (not just linear ones).

To incorporate the concept of nonlinearity, it's effective to apply a nonlinear function, called the *activation function*, to each neuron's output. Three of the most commonly used activation functions are *sigmoid* (sig), *hyperbolic tangent* (tanh), and the *ramp* function (ReLu) plotted in figure 8.2.

You don't have to worry too much about which activation function is better under what circumstances. That's still a pretty active research topic. Feel free to experiment with the three shown in figure 8.2. Usually, the best one is chosen through cross-validation.

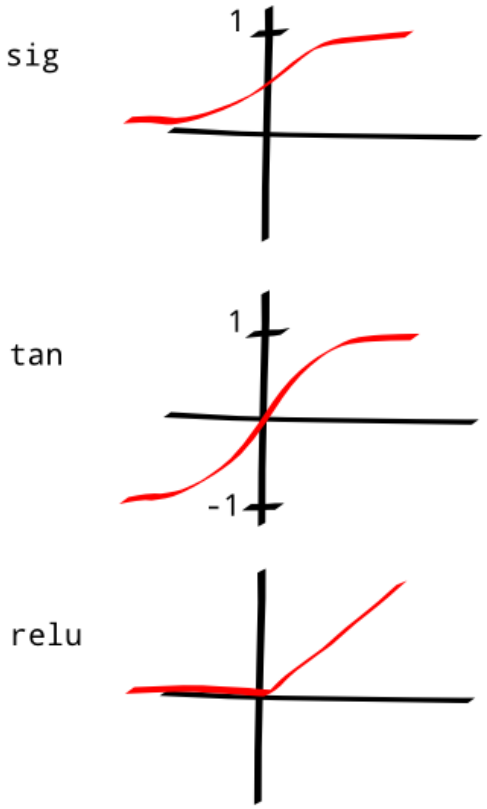


Figure 8.2 We use non-linear functions such as sig, tan, and relu to introduce non-linearity to our models.

The sigmoid function isn't new to us. If you recall, the logistic regression classifier in Chapter 4 applied this sigmoid function to a linear function $w * x + b$. The neural network model in figure

8.3 represents the function *f(x) = sig(w * x + b)*. It is a 1-input and 1-output network, where *w* and *b* are the parameters of this model.

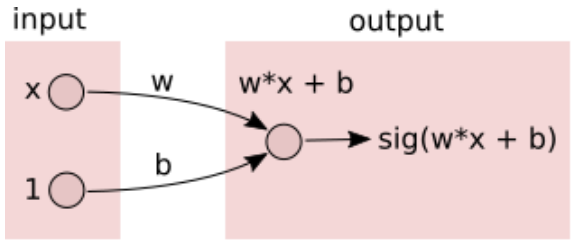


Figure 8.3 A non-linear function, such as sigmoid, is applied to the output of a node

If instead we have two inputs (*x1* and *x2*), we can modify our neural network to look like the one in figure 8.4. Given training data and a cost function, the parameters to be learned are *w1*, *w2*, and *b*. When trying to model data, having multiple inputs to a function is very common. For example, image classification takes the entire image (pixel-by-pixel) as the input.

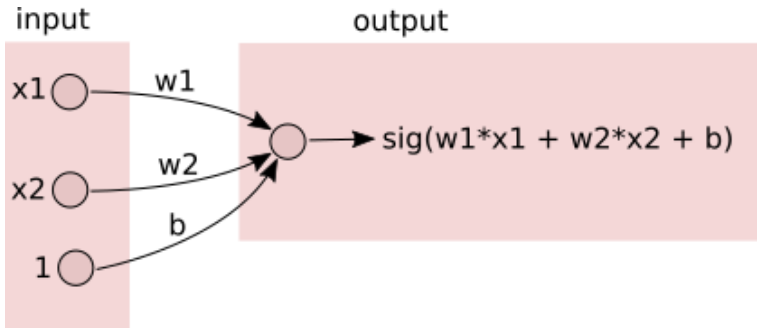


Figure 8.4 A 2-input network will have 3 parameters (w1, w2, and b).

Naturally, we can generalize to an arbitrary number of inputs (*x1*, *x2*, …, *xn*). The corresponding neural network represents the function

$$f(x1, ..., xn) = sig(wn*xn + ... + w1*x1 + b)$$

as seen in figure 8.5.

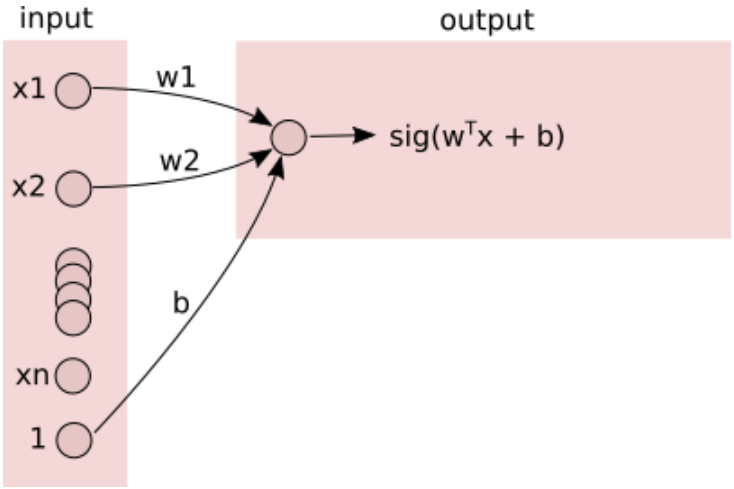


Figure 8.5 The input dimension can be arbitrarily long.

So far, we've only dealt with an input layer and an output layer. Nothing's stopping us from arbitrarily adding neurons in-between. Neurons that are neither used as input nor output are called hidden neurons.

Turns out, as long as the activation function is something nonlinear, a neural network with at least one hidden layer can approximate arbitrary functions. In linear models, no matter what parameters are learned, the function remains linear. The neural network model, on the other hand, is flexible enough to approximately represent any function! What a time to be alive!

A hidden layer is any collection of hidden neurons that do not connect to each other, as seen in figure 8.6. Adding more hidden layers greatly improves the expressive power of the network.

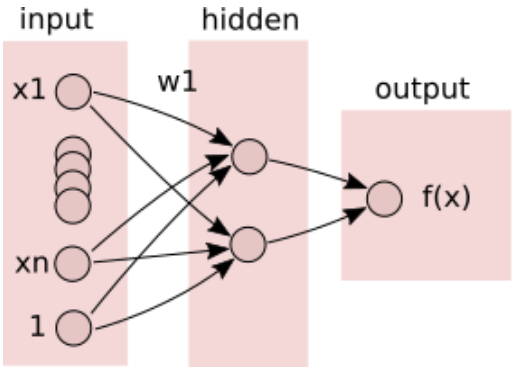


Figure 8.6 Nodes that interface with neither the input nor output are called hidden units. A hidden layer is a collection of hidden units that are not connected to each other.

TensorFlow comes with many helper functions to help you obtain the parameters of a neural network in an efficient way. We'll see how to invoke those tools in this chapter when we start using our very first neural network architecture called autoencoders.

8.2 Autoencoder

An autoencoder is a type of neural network that tries to learn parameters that make the output as close to the input as possible. An obvious way to do so is simply return the input directly as shown in figure 8.7.

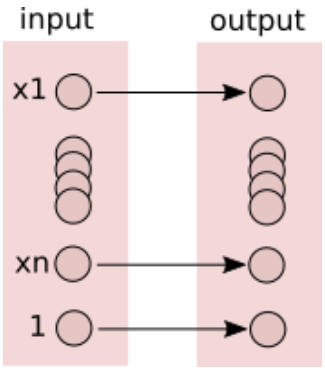


Figure 8.7 If we want to create a network where the input equals the output, we can just connect the corresponding nodes and set each parameter's weight to 1.

But an autoencoder is more interesting than that. It contains a hidden layer! If that hidden layer has a smaller dimension than the input, the hidden layer is a compression of your data, called *encoding*. The process of reconstructing the input from the hidden layer is called *decoding*. Figure 8.8 shows an example of an autoencoder.

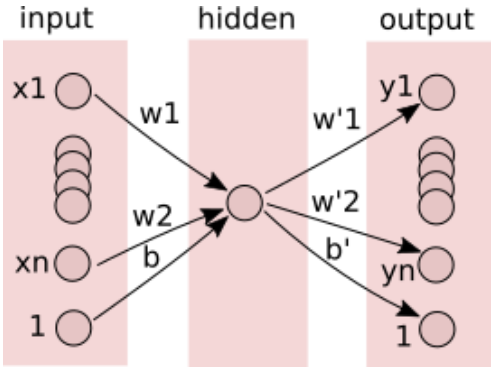


Figure 8.8 Here, we introduce a restriction to a network that tries to reconstruct its input. Data will pass through a narrow channel, as seen by the hidden layer. In this example, there is only 1 node in the hidden layer. So, this network is trying to encode (and decode) an n-dimensional input signal into just 1

dimension, which will likely be very difficult in practice.

Encoding is a great way to reduce the dimension of the input. For example, if we can represent a 256 by 256 image in just 100 hidden nodes, then we've reduced each data item by a factor of hundreds!

It makes sense to use an object-oriented programming style to implement an autoencoder. That way, we can later reuse the class in other applications without worrying about tightly coupled code. In fact, creating our code as outlined in listing 8.1 helps build deeper architectures, such as a *stacked autoencoder*, which performs better empirically.

TIP Generally with neural networks, adding more hidden layers seems to improve performance if you have enough data to not overfit the model.

Listing 8.1 Python class schema

```
class Autoencoder:
    def __init__(self, input_dim, hidden_dim): ❶

    def train(self, data): ❷

    def test(self, data):  ❸
```

❶ Initialize variables
❷ Train on a dataset
❸ Test on some new data

Let's open a new Python source file and call it `autoencoder.py`. This file will store the autoencoder class that we'll use from a separate piece of code.

The constructor will set up all the TensorFlow variables, placeholders, optimizers, and operators. Anything that doesn't immediately need a session can go in the constructor. Because we're dealing with two sets of weights and biases (one for the encoding step and the other for the decoding), we can use TensorFlow's name scopes to disambiguate a variable's name.

For instance, listing 8.2 shows an example of defining a variable within a named scope. Now we can seamlessly save and restore this variable without worrying about name-collisions.

Listing 8.2 Using name scopes

```
with tf.name_scope('encode'):
    weights = tf.Variable(tf.random_normal([input_dim, hidden_dim], dtype=tf.float32),
       name='weights')
    biases = tf.Variable(tf.zeros([hidden_dim]), name='biases')
```

Moving on, let's implement the constructor as shown in listing 8.3.

Listing 8.3 Autoencoder class

```
import tensorflow as tf
import numpy as np

class Autoencoder:
  def __init__(self, input_dim, hidden_dim, epoch=250, learning_rate=0.001):
    self.epoch = epoch (1)
    self.learning_rate = learning_rate (2)

    x = tf.placeholder(dtype=tf.float32, shape=[None, input_dim]) (3)

    with tf.name_scope('encode'): (4)
       weights = tf.Variable(tf.random_normal([input_dim, hidden_dim], dtype=tf.float32),
       name='weights')
       biases = tf.Variable(tf.zeros([hidden_dim]), name='biases')
       encoded = tf.nn.tanh(tf.matmul(x, weights) + biases)
       with tf.name_scope('decode'): (5)
       weights = tf.Variable(tf.random_normal([hidden_dim, input_dim], dtype=tf.float32),
       name='weights')
       biases = tf.Variable(tf.zeros([input_dim]), name='biases')
       decoded = tf.matmul(encoded, weights) + biases

    self.x = x  (6)
    self.encoded = encoded  (6)
    self.decoded = decoded  (6)

    self.loss = tf.sqrt(tf.reduce_mean(tf.square(tf.sub(self.x, self.decoded)))) (7)
    self.train_op = tf.train.AdamOptimizer(self.learning_rate).minimize(self.loss) (8)
    self.saver = tf.train.Saver()  (9)
```

(1) Number of learning cycles
(2) Hyper-parameter of optimizer
(3) Define the input layer dataset
(4) Define the weights and biases under a name scope so we can tell them apart from the decoder's weights and biases
(5) The decoder's weights and biases are defined under this name scope
(6) These will be method variables
(7) Define the reconstruction cost
(8) Choose the optimizer
(9) Setup a saver to save model parameters as they're being learned

Now, in listing 8.3, we'll define a class method called `train` that will receive a dataset and learn parameters to minimize its loss.

Listing 8.3 Train the autoencoder

```
    def train(self, data):
    num_samples = len(data)
    with tf.Session() as sess:       (1)
        sess.run(tf.initialize_all_variables())  (1)
        for i in range(self.epoch):  (2)
            for j in range(num_samples):         (3)
              l, _ = sess.run([self.loss, self.train_op], feed_dict={self.x: [data[j]]})  (3)
                if i % 10 == 0:      (4)
```

```
                print('epoch {0}: loss = {1}'.format(i, l)) 4
                self.saver.save(sess, './model.ckpt')       5
        self.saver.save(sess, './model.ckpt')               5
```

1. Start a TensorFlow session and initialize all variables
2. Iterate through the number of cycles defined in the constructor
3. One-by-one train the neural network on a data item
4. Print the reconstruction error once every 10 cycles
5. Save the learned parameters to file

You now have enough code to design an algorithm that learns an autoencoder from arbitrary data. Before we start using this class, let's create one more method. As shown in listing 8.4, the test method will let you evaluate the autoencoder on new data.

Listing 8.4 Test the model on some data

```
def test(self, data):
    with tf.Session() as sess:
       self.saver.restore(sess, './model.ckpt')  1
       hidden, reconstructed = sess.run([self.encoded, self.decoded], feed_dict={self.x:
       data})  2
    print('input', data)
    print('compressed', hidden)
    print('reconstructed', reconstructed)
    return reconstructed
```

1. Load the learned parameters
2. Reconstruct the input

Finally, let's create a new Python source file called `main.py` and use our Autoencoder class, as show in listing 8.5.

Listing 8.5 Using our autoencoder class

```
from autoencoder import Autoencoder
from sklearn import datasets

hidden_dim = 1
data = datasets.load_iris().data
input_dim = len(data[0])
ae = Autoencoder(input_dim, hidden_dim)
ae.train(data)
ae.test([[8, 4, 6, 2]])
```

8.3 Batch training

Training a network one-by-one is the safest bet if you're not pressured with time. But if your network is taking longer than desired, one solution is to train it with multiple data inputs at a time, called batch training.

Typically, as the batch size increases, the algorithm speeds up, but has less guarantees of successful convergence. It's a double-edged sword. Go wield it in listing 8.6. We'll use that helper function later.

Listing 8.6 Batch helper function

```
def get_batch(X, size):
   a = np.random.choice(len(X), size, replace=False)
   return X[a]
```

To use batch learning, you'll need to modify the `train` method from listing 8.3. The batch version is shown in listing 8.7. It inserts an additional inner loop for each batch of data. Typically, the number of batch iterations should be enough so that all data is covered in the same epoch.

Listing 8.7 Batch learning

```
def train(self, data, batch_size=10):
    with tf.Session() as sess:
       sess.run(tf.initialize_all_variables())
       for i in range(self.epoch):
       for j in range(500):  (1)
               batch_data = get_batch(data, self.batch_size) (2)
               l, _ = sess.run([self.loss, self.train_op], feed_dict={self.x: batch_data})
               if i % 10 == 0:
                 print('epoch {0}: loss = {1}'.format(i, l))
                 self.saver.save(sess, './model.ckpt')
       self.saver.save(sess, './model.ckpt')
```

(1) Loop through various batch selections
(2) Run the optimizer on a randomly selected batch

8.4 Working with images

Most neural networks, like our autoencoder, only accept one-dimensional input. Pixels of an image, on the other hand, are indexed by both rows and columns. Moreover, if a pixel is in color, it has a value for its red, green, and blue concentration, as seen in figure 8.9.

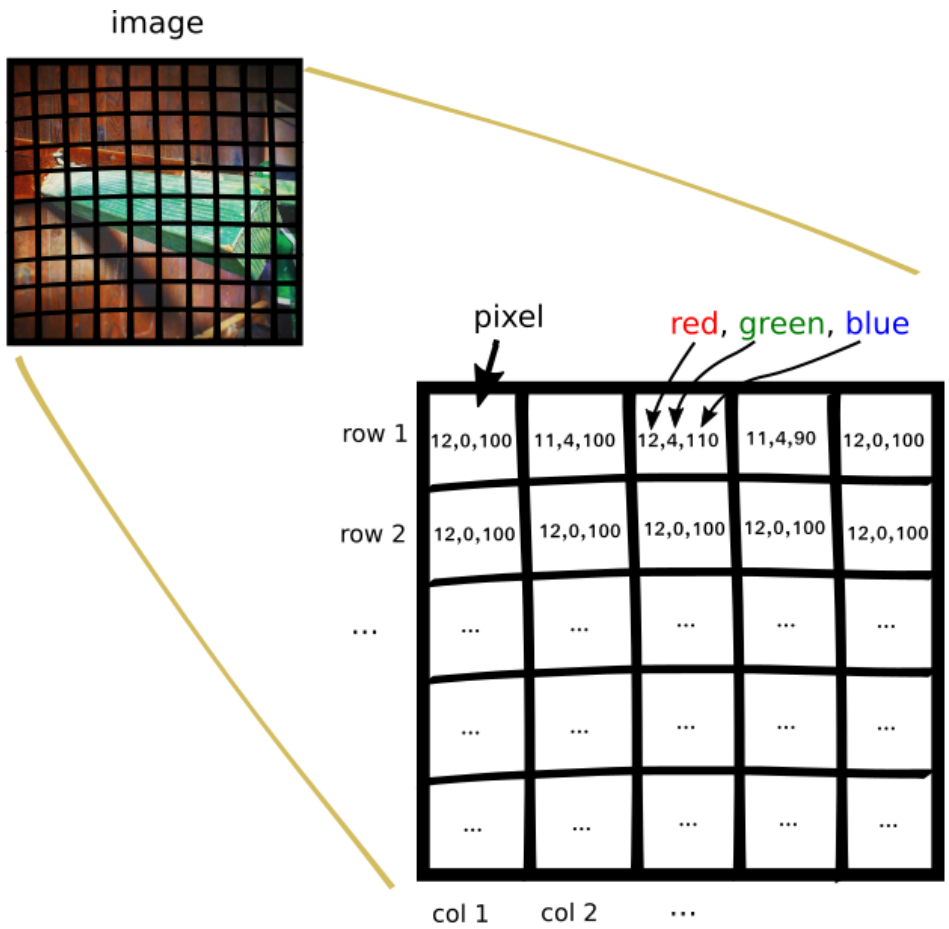


Figure 8.9 A colored image is composed of pixels, and each pixel contains a red, green, and blue value.

A convenient way to manage the higher dimensions of an image involves two steps:

1. Convert the image to grayscale: merge the values of red, green, and blue into what is called the *pixel intensity*, which is a weighted average of the color values.
2. Rearrange the image into row-major order. That way we can index the image by 1 number instead of 2. If an image is 3 by 3 pixels in size, we rearrange into the structure shown in figure 8.10.

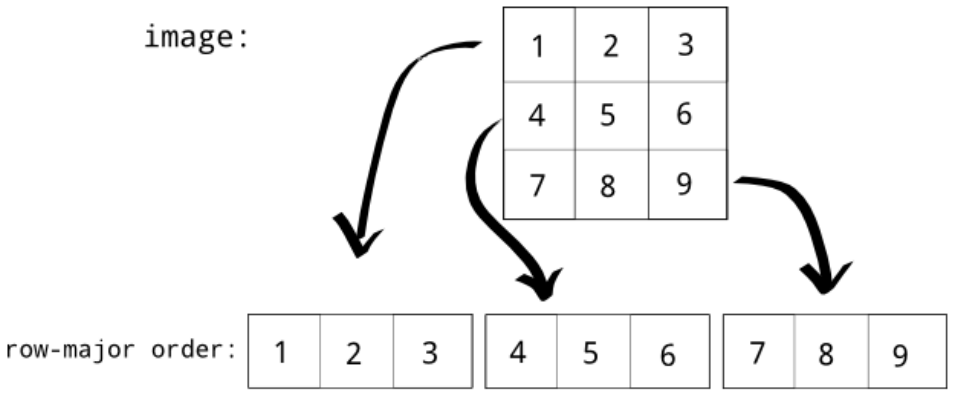


Figure 8.10 An image can be represented in row-major order. That way, we can represent a 2-dimensional structure as a 1-dimensional structure.

There are many ways to use images in TensorFlow. If you have pictures laying around on your hard drive, you can load them using `scipy`, which comes with TensorFlow. The code in listing 8.8 shows you how to load an image as in grayscale, resize it, and represent it in row-major order.

Listing 8.8 Loading images

```
from scipy.misc import imread, imresize

gray_image = imread(filepath, True) ❶
small_gray_image = imresize(gray_image, 1. / 8.) ❷
x = small_gray_image.flatten()       ❸
```

❶ load an image as grayscale
❷ resize it to something smaller
❸ convert it to a 1 dimensional structure

Image processing is a lively field of research, so datasets are readily available for us to use, instead of using our own limited images. For instance, there's a dataset called CIFAR-10, which contains 60,000 labeled images, each 32 by 32 in size.

Download the python dataset from https://www.cs.toronto.edu/~kriz/cifar.html. Place the extracted `cifar-10-batches-py` folder in your working directory. The code in listing 8.9 is provided from the CIFAR webpage. In a new file, called `main2.py`, follow along to the listings.

Listing 8.9 Reading from extracted CIFAR-10 dataset

```
import cPickle

def unpickle(file):  ❶
    fo = open(file, 'rb')
    dict = cPickle.load(fo)
```

```
    fo.close()
    return dict
```

❶ Reads the CIFAR file, returning the loaded dictionary

Let's read each of the dataset files using the `unpickle` function we just created. The CIFAR dataset contains 6 files, each prefixed with "data_batch_" and followed by a number. Each file contains information about the image data and corresponding label. Listing 8.10 shows how to loop through all the files and append the datasets to memory.

Listing 8.10 Reading all CIFAR-10 files to memory

```
import numpy as np

names = unpickle('./cifar-10-batches-py/batches.meta')['label_names']
data, labels = [], []
for i in range(1, 6):  ❶
    filename = './cifar-10-batches-py/data_batch_' + str(i)
    batch_data = unpickle(filename)  ❷
    if len(data) > 0:
        data = np.vstack((data, batch_data['data']))
        labels = np.vstack((labels, batch_data['labels']))
    else:
        data = batch_data['data']
        labels = batch_data['labels']
```

❶ Loop through the 6 files
❷ Load the file to obtain a Python dictionary

Each image is represented as a series of red pixels, followed by green pixels, and then blue pixels. Listing 8.11 will create a helper function to convert the image into grayscale by simply averaging the red, green, and blue values.

> **BY THE WAY** There are other ways to achieve more realistic grayscale, but this approach of simply averaging the 3 values gets the job done as well. Human perception is more sensitive to green light, so in some other versions of gray-scaling, green values might play a higher weight in the averaging.

Listing 8.11 Converting CIFAR-10 image to grayscale

```
def grayscale(a):
    return a.reshape(a.shape[0], 3, 32, 32).mean(1).reshape(a.shape[0], -1)

data = grayscale(data)
```

Lastly, let's collect all images of a certain class, such as "horse." We'll run our autoencoder on all pictures of horses as show in listing 8.12.

Listing 8.12

```
from autoencoder import Autoencoder

x = np.matrix(data)
y = np.array(labels)

horse_indices = np.where(y == 7)[0]

horse_x = x[horse_indices]

print(np.shape(horse_x))  # (5000, 3072)

input_dim = np.shape(horse_x)[1]
hidden_dim = 100
ae = Autoencoder(input_dim, hidden_dim)
ae.train(horse_x)
```

Congratulations! You can now encode images similar to your training dataset into just 100 numbers. This autoencoder model is one of the simplest, so clearly it will be a lossy encoding.

8.5 Modern Autoencoders

This chapter introduced the most straightforward type of autoencoder, but other variants have been studied, each with their benefits and applications. Let's take a look at a couple.

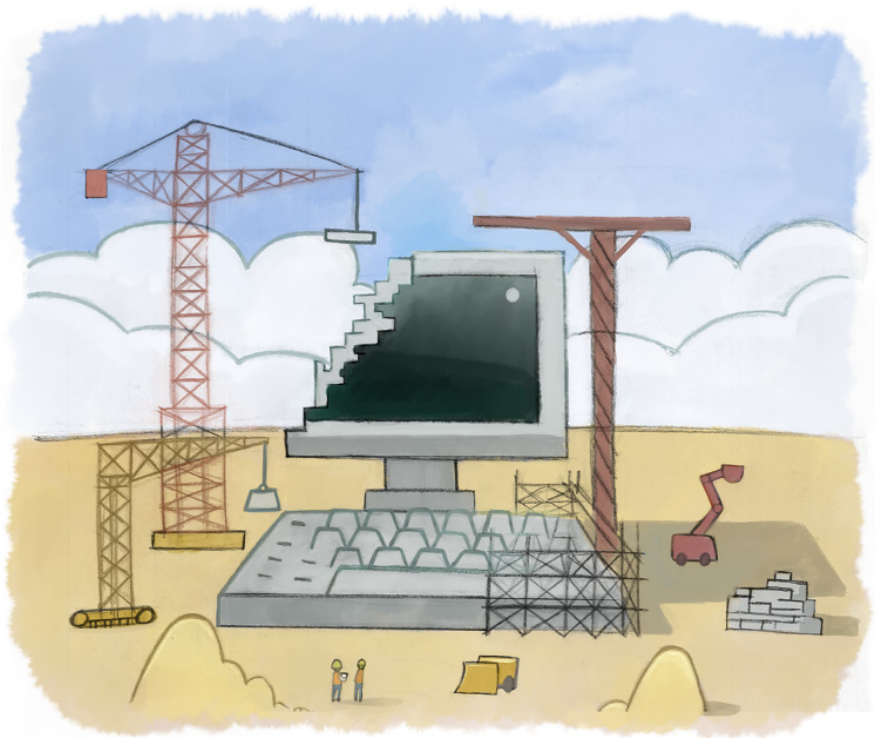
- A *stacked autoencoder* starts the same way a normal autoencoder does. It learns the encoding for an input into a smaller hidden layer by minimizing the reconstruction error. The hidden layer is now treated as the input to a new autoencoder that tries to encode the first layer of hidden neurons to an even smaller layer (the second layer of hidden neurons). This continues as desired. Often, the learned encoding weights are used as initial values for solving regression or classification problems in a deep neural network architecture.
- A denoising autoencoder receives a noised up input instead of the original input, and it tries to "denoise" it. The cost function is no longer to minimize the reconstruction error. Now, we are trying to minimize the error between the denoised image and the original image. The intuition is that our human minds can still comprehend a photograph even after scratches or markings over it. If a machine can also see through the noised input to recover the original data, maybe it has a better understanding of the data. Denoising models have shown to better capture salient features on an image.
- A variational autoencoder can generate new natural images given the hidden variables directly. Let's say you encode a picture of a man as a 100-dimensional vector, and then a picture of a woman as another 100-dimensional vector. You can take the average of the two vectors, run it through the decoder, and produce a reasonable image that represents visually a person that is between a man and woman. This generative power of the variational autoencoder is derived from a type of probabilistic models called Bayesian networks.

8.6 Summary

- A neural network is useful when a linear model is ineffective to describe the dataset.
- Autoencoders are unsupervised learning algorithms that try to reproduce their inputs, and in doing so reveal interesting structure about the data.
- Image can easily be fed as input to a neural network by flatting and grayscaling.

A

Software installation



This appendix covers

- Installing TensorFlow
- Installing Matplotlib

You can install TensorFlow in a couple of ways. If you're familiar with UNIX based systems (like Linux or OSX), feel free to use one of the approaches on the official documentation: https://www.tensorflow.org/get_started/os_setup.html. In this appendix, we will cover one of these installation methods that works on all platforms, including Windows. Specifically, let's install TensorFlow using a Docker container.

A.1 Installing TensorFlow using Docker

Docker is a system for packaging a software's dependencies to keep everyone's installation environment identical. This standardization helps limit inconsistencies between different computers. It's a relatively recent technology, so let's go through how to use it in this appendix.

> **TIP** There are many ways to install TensorFlow other than using a Docker container. Visit the official documentations for more details on how to install TensorFlow: https://www.tensorflow.org/get_started/os_setup.html

A.1.1 Install Docker on Windows

Docker only works on 64-bit windows (7 or above) with virtualization enabled. Fortunately, most consumer laptops and desktops easily satisfy this requirement. To check whether your computer supports Docker, open up `Control Panel`, click `System and Security`, and then click `System`. Here you can see the details about your Windows machine, including processor and system type. If the system is 64-bit then you're almost good to go.

The next step is to check if your processor can support virtualization. On windows 8 or above, you can simply open up the `Task Manager` (`Ctrl + Shirt + Esc`) and click on the `Performance` tab. If "Virtualization" shows up as "Enabled" then you're all set. See figure A.1 for reference. For Windows 7, you should use the Microsoft Hardware-Assisted Virtualization Detection Tool (https://www.microsoft.com/en-us/download/details.aspx?id=592).

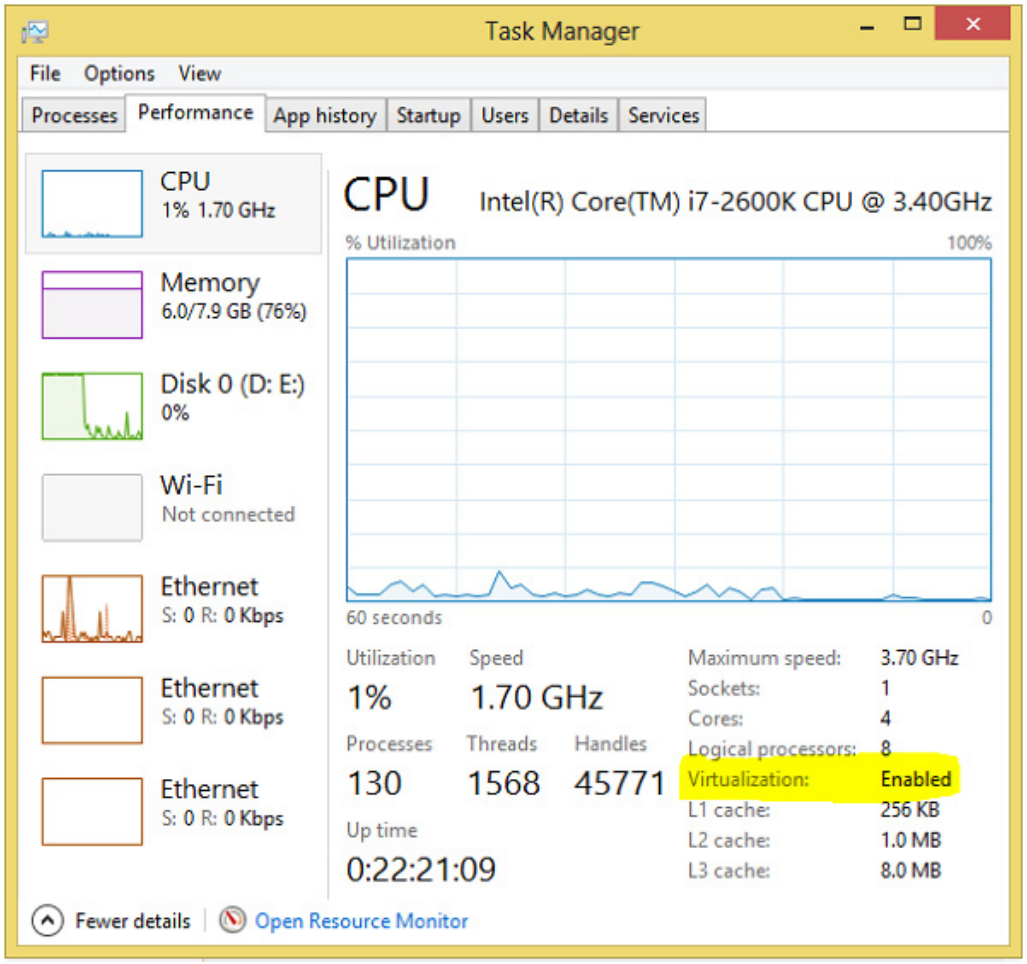


Figure A.1 Ensure your 64-bit computer has virtualization enabled.

Now that you know if your computer can support Docker, let's install the Docker Toolbox located at https://www.docker.com/products/docker-toolbox. Run the downloaded setup executable and accept all the defaults by pressing next on the dialog boxes. Once installed, run the `Docker Quickstart Terminal`.

A.1.2 Install Docker on Linux

Docker is officially supported on a number of Linux distributions. Namely, the official Docker documentation (https://docs.docker.com/engine/installation/linux/) contains tutorials for Arch Linux, CentOS, CRUS Linux, Debian, Fedora, FrugalWare, Gentoo, Oracle Linux, Red Hat

Enterprise Linux, openSUSE, and Ubuntu. Docker is native to Linux so there is typically no problem installing it.

A.1.3 Install Docker on OSX

Docker works on OSX 10.8 "Mountain Lion" or newer. Install the Docker Toolbox from https://www.docker.com/products/docker-toolbox. After installation, open the "Docker Quickstart Terminal" from the Applications folder or the Launchpad.

A.1.4 How to user Docker

Install VirtualBox from https://www.virtualbox.org/wiki/Downloads. You'll need it later to manage your Docker container.

From the `Docker Quickstart Terminal`, run the following command to create a new container called `vdocker`.

```
$ docker-machine create vdocker -d virtualbox
```

Next, launch the TensorFlow binary image using the following command, as shown in Figure A.2.

```
$ docker run -it b.gcr.io/tensorflow/tensorflow
```

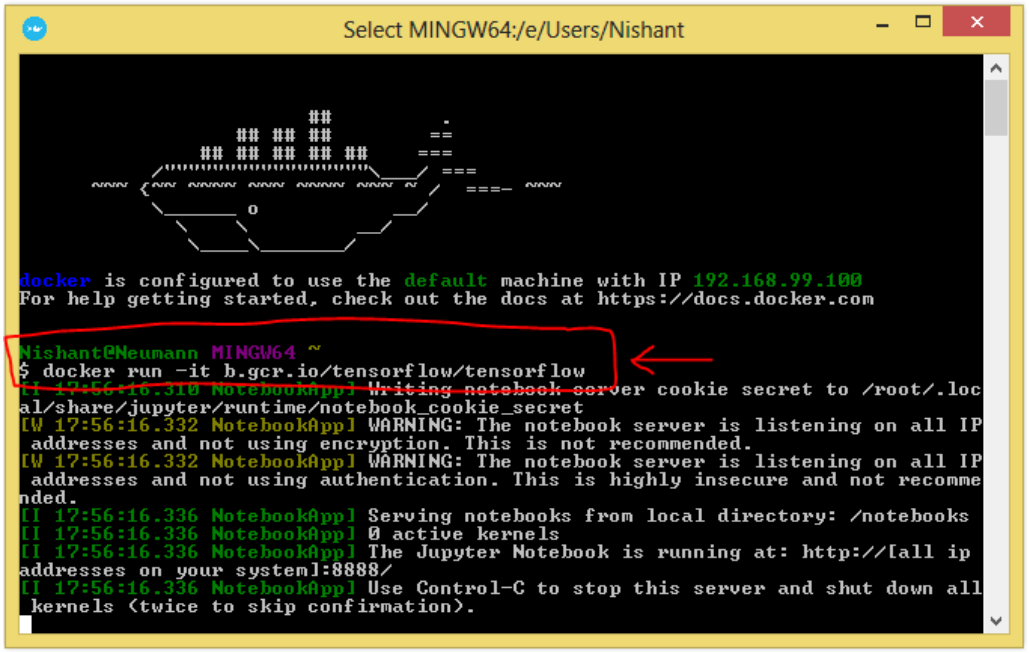


Figure A.2 Run the official TensorFlow

Open VirtualBox, and you'll find the `vdocker` machine status as shown in Figure A.3.

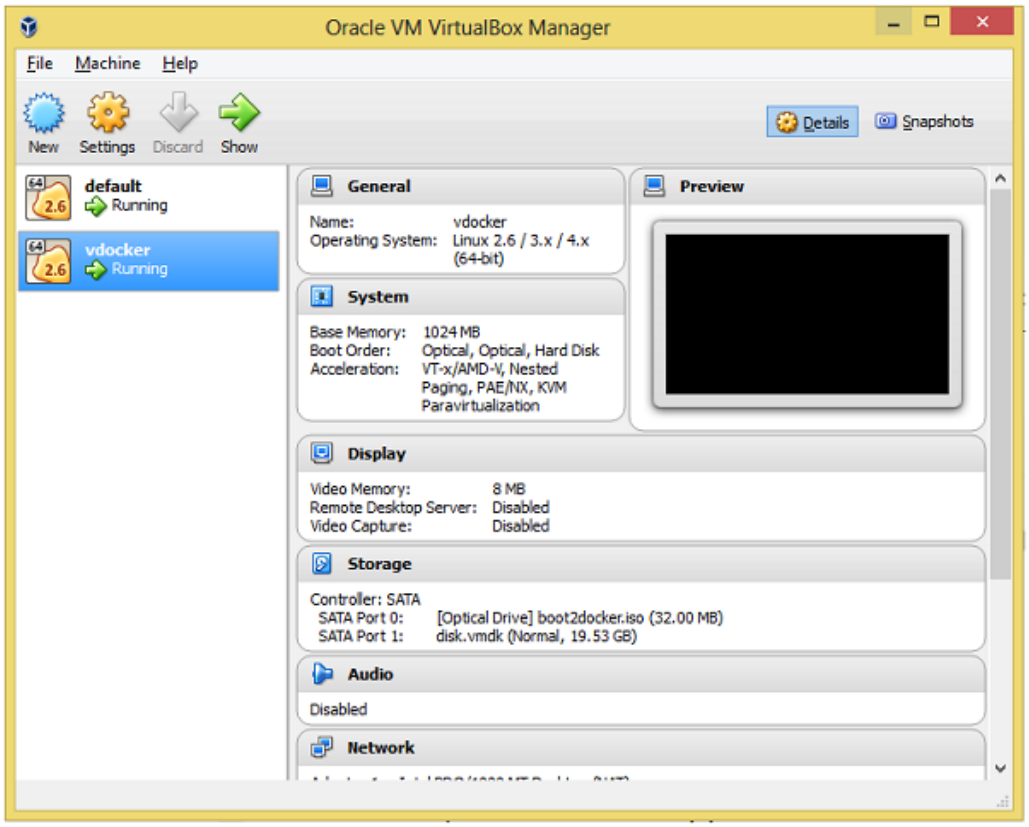


Figure A.3 VirtualBox shows the available Docker containers.

Click `Settings`, navigate to `Network`, and add a new port forwarding rule, as shown in Figure A.4.

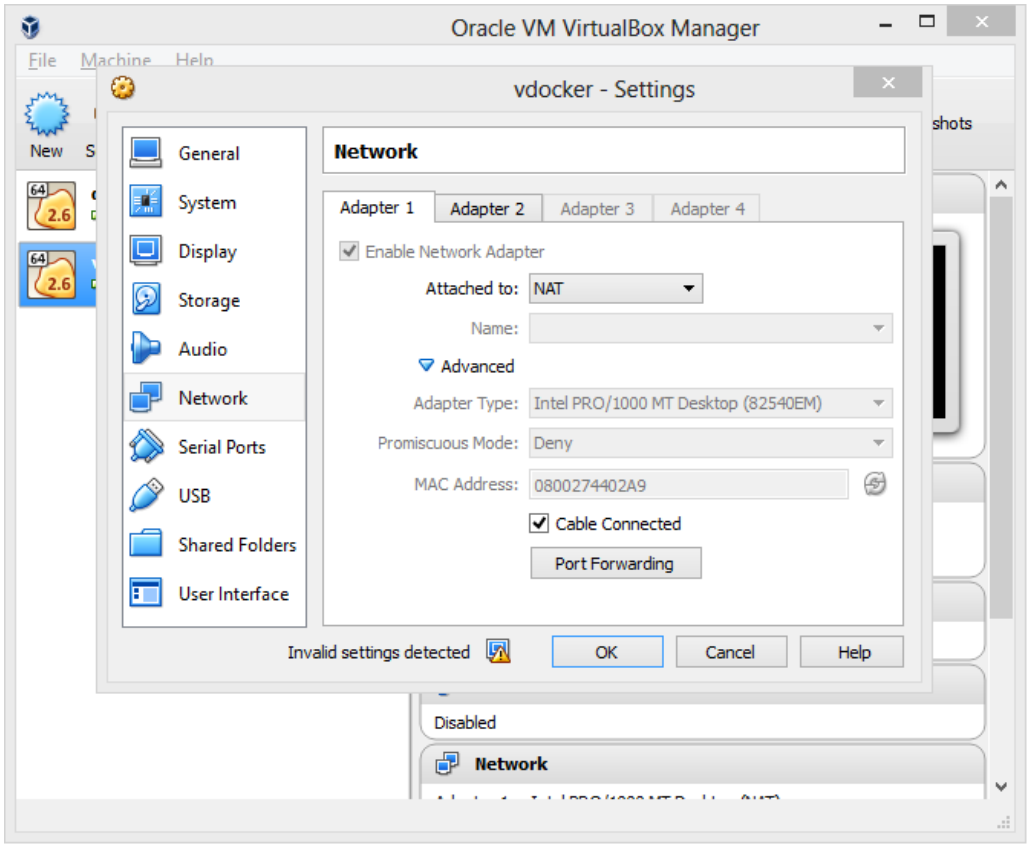


Figure A.4 The Settings menu reveals the customization options. Select Network to add a port forwarding rule.

Set its host port to *8810* and its guest port to *8888*. `OK` to close out of the windows. See Figure A.5.

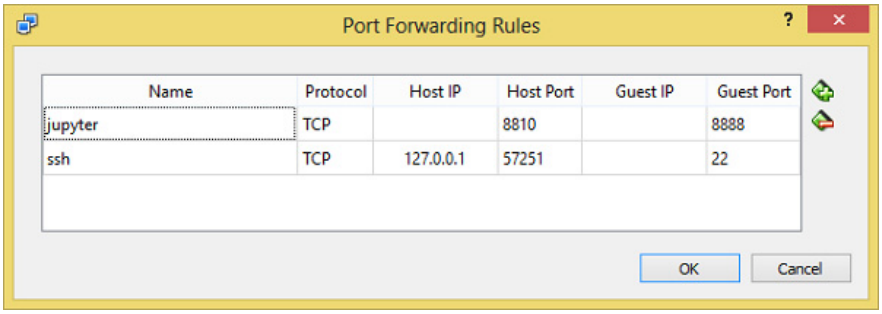


Figure A.5 Add a new port rule to be able to use Jupyter, which is an interactive Python interface.

Open a browser and navigate to http://localhost:8810. Figure A.6 shows the Jupyter notebook accessed through a browser.

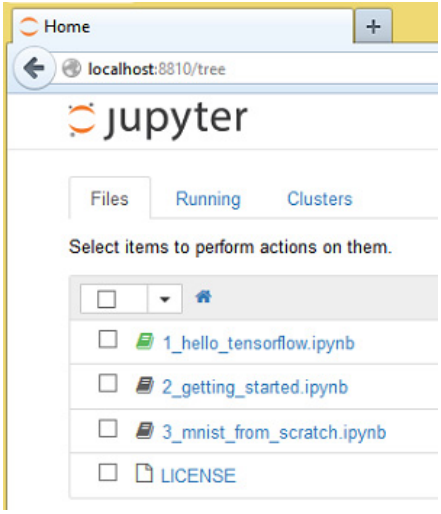


Figure A.6 We can interact with TensorFlow through a Python interface called Jupyter.

A.2 Installing Matplotlib

Matplotlib is a cross-platform Python library for plotting 2D visualizations of data. As a general rule of thumb, if your computer can successfully run TensorFlow, then it will have no trouble installing Matplotlib. Install it by following the official documentation on http://matplotlib.org/users/installing.html.