



MACHINE LEARNING

WITH
TensorFlow

Nishant Shukla

MEAP

 MANNING



**MEAP Edition
Manning Early Access Program
Machine Learning with TensorFlow
Version 10**

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to

www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/machine-learning-with-tensorflow>

Licensed to Shashank Nainwal <hybridboy11@gmail.com>

welcome

Dear fellow early adopters, curious readers, and puzzled newcomers,

Thank you all for every bit of communication with me, whether it be through the official book forums, through email, on GitHub, or even on Reddit. I've listened carefully to your questions, suggestions, and concerns, regardless of whether or not I've replied to you (and I do apologize for not replying to you).

In the latest edition, I am proud to announce a beautiful makeover of every chapter. The text is greatly improved and slowed down to better cover complex matters, especially the areas where you requested more explanation. Most figures and mathematical equations have been updated to look crisp and professional. The code is now updated to TensorFlow v1.0, and it is also available on GitHub at <https://github.com/BinRoot/TensorFlow-Book/>. Also, the chapters are rearranged to better deliver the right skills at the right time, if the book were read in order.

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 just over 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, mistakes, and improvements because 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

PART 1 MY MACHINE LEARNING RIG

1 A machine learning odyssey

2 TensorFlow essentials

PART 2 CORE LEARNING ALGORITHMS

3 Linear regression and beyond

4 A gentle introduction to classification

5 Automatically clustering data

6 Hidden Markov models

PART 3 THE NEURAL NETWORK PARADIGM

7 A peek into autoencoders

8 Reinforcement learning

9 Convolutional neural networks

10 Recurrent neural networks

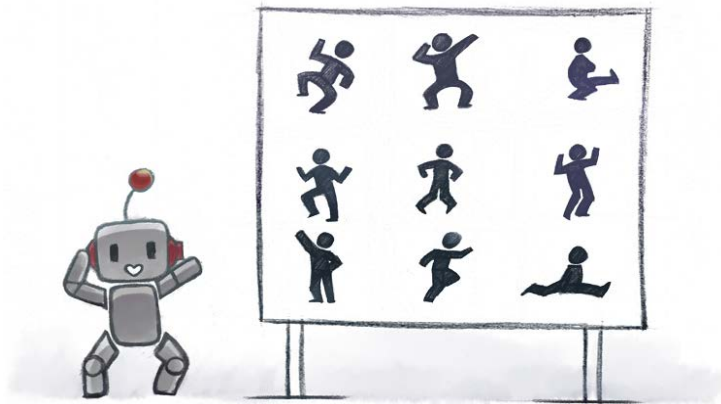
11 Sequence-to-sequence models for chatbots

12 Utility landscape

APPENDIX

A Installation

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).

As the study of ML matures, the tools have become more standardized, robust, performant, and scalable. This is where TensorFlow comes in. It’s a software library with an intuitive interface that lets programmers dive into using complex ML ideas. The next chapter will go through the ins and outs of this library, and every chapter thereafter will explain how to use TensorFlow for each of the various ML applications.

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
$x_1 = (2, 2)$	$y_1 = \text{Even}$
$x_2 = (3, 2)$	$y_2 = \text{Odd}$
$x_3 = (2, 3)$	$y_3 = \text{Odd}$
$x_4 = (3, 3)$	$y_4 = \text{Even}$
...	...

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	$2m + 2n = 2(m + n)$ <p>even</p>	$2m + (2n+1) = 2m + 2n + 1$ <p>odd</p>
odd	$(2m+1) + 2n = 2m + 2n + 1$ <p>odd</p>	$(2m+1) + (2n+1) = 2(m + n + 1)$ <p>even</p>

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.

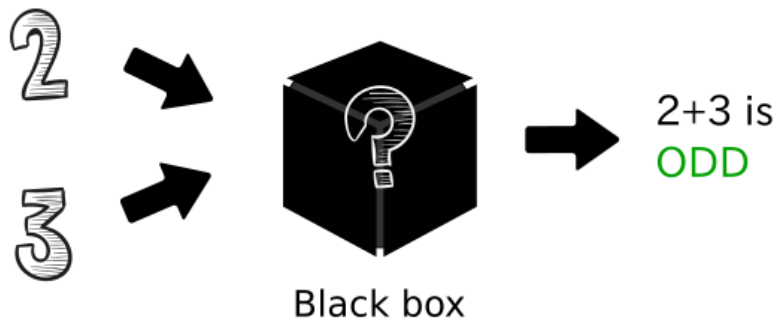


Figure 1.3 An ML approach to solving problems can be thought of as tuning the parameters of a black box until it produces satisfactory results.

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 called the *feature vector*, and summarize it into a *model*. The model is our recipe. In effect, the model is a program with a couple of open interpretations, and the data helps disambiguate it.

WHAT IS A FEATURE VECTOR? A feature vector is a practical simplification of data. You can think of it as a sufficient summary of real-world objects into a list of attributes. The learning and inference steps rely on the feature vector instead of the data directly.

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.

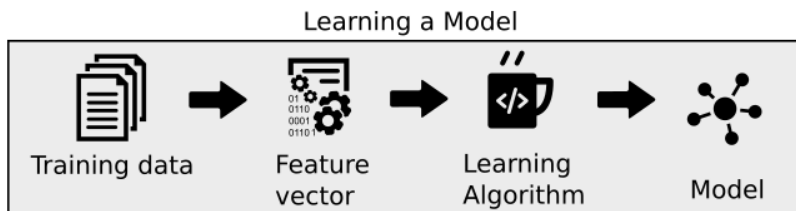


Figure 1.4 The general learning approach follows a structured recipe. First, the dataset needs to be transformed into a representation, most often a list of vectors, which can be used by the learning algorithm. The learning algorithm chooses a model and efficiently searches for the model's parameters.

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.

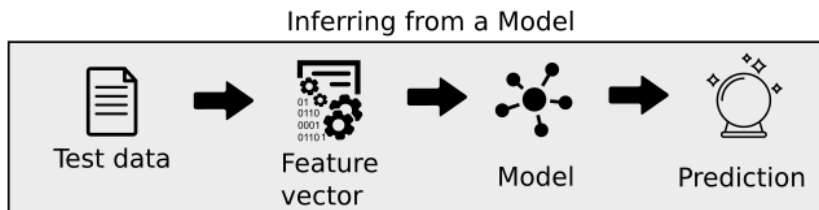


Figure 1.5 The general inference approach uses a model that has already been either learned or simply given. After converting data to a usable representation, such as a feature vector, it uses the model to produce intended

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.

The basic theme in all forms of representation is the idea of *features*, which are observable properties of an object.

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 , $\pi/2$, $\pi/3$). You get the idea, just a collection of numbers of the same type. In a machine learning program, a vector measures a property of the data, such as color, density, loudness, or proximity – anything you can describe with a series of numbers, one for each thing being measured.
2. Moreover, a vector of vectors is a *matrix*. If each feature vector describes the features of one object in your data set, the matrix describes all the objects – each item in the outer vector is a node that's a list of features of one object.
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.

Feature vectors are a practical simplification of real-world data, which can be too complicated to deal with in the real world. Instead of attending to every little detail of a data item, a feature vector is a practical simplification. For example, a car in the real-world is much more than the text used to describe it. A car salesman is trying to sell you the car, not the intangible words spoken or written. Those words are just abstract concepts, very similar to how feature vectors are just summaries of the data.

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. This ordered list of features is the 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, or you'll be losing information you care about, and not too many, or it will be unwieldy and time-consuming to keep track of. 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.

For example, when training a model to detect cars in an image, you will gain an enormous performance and speed improvement if you first convert the image to grayscale. By providing some of your own bias into the preprocessing of the data, you end up helping the algorithm because it will not need to learn that colors don't quite matter when detecting cars. The algorithm can instead focus on identifying shapes and textures, which may will lead to much faster learning than trying to process colors as well.

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.

Curse of dimensionality

In order to accurately model real-world data, we clearly need more than just 1 or 2 data points. But just how much data depends on a variety of things, including the number of dimensions in the feature vector. Adding too many features causes the number of data points required to describe the space to increase exponentially. That's why you can't just design a 1,000,000 dimensional feature vector to exhaust all possible factors and then expect the algorithm to learn a model. This phenomena is called the curse of dimensionality.



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 is 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. We'd need to add another feature to the system in order to tell them apart, or we think they are the same item. 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. (Hint: what types of words do retailers use to describe their clothing online?)



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.

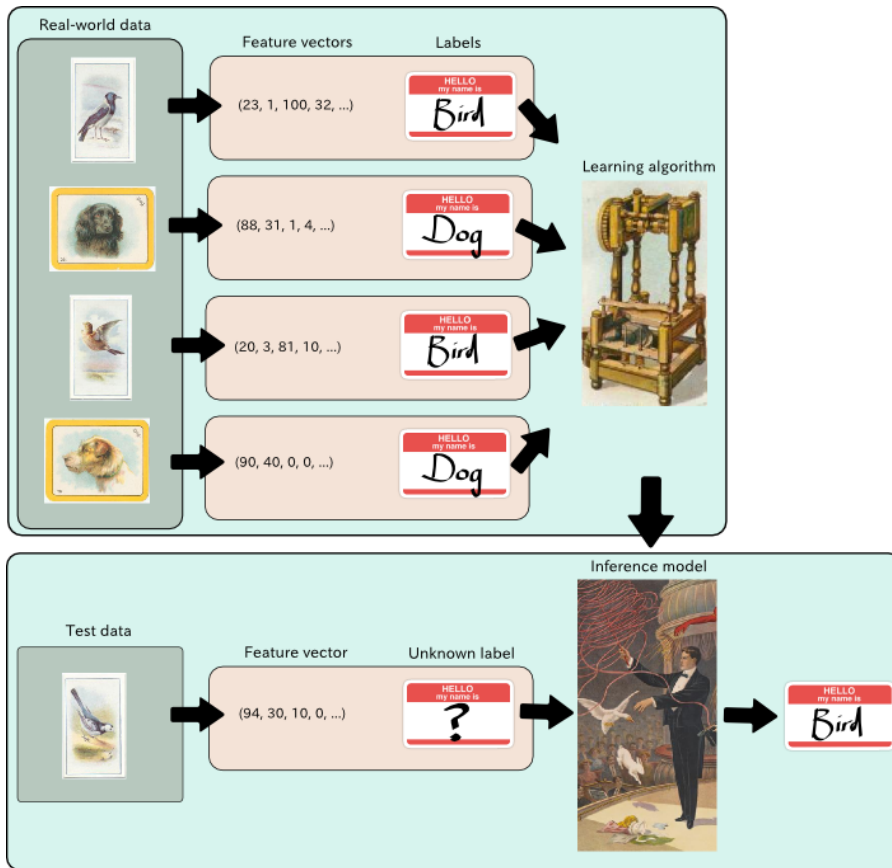


Figure 1.9 Feature vectors are a representation of real world data used by both the learning and inference components of machine learning. The input to the algorithm is not the real-world image directly, but instead its feature vector.

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, $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$. The Euclidian distance $||x-y||$ is calculated by

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$

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

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

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 L0 distance between (1,1) and (2,2) is 2, because neither dimension matches up. Imagine if the first and second dimensions represent username and password, respectively. If the L0 distance between a login attempt and the true credentials is 0, then the login is successful. If the distance is 1, then either the username or password is incorrect, but not both. Lastly if the distance is 2, both username and password are not found in the database.
- The *L1 norm* is defined as $\sum |x_n|$. 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. Computing the L1 distance between two vectors is essentially the sum of absolute differences at each dimension, which is a useful measure of similarity.

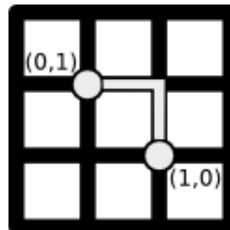


Figure 1.10 The L1 distance is also called the taxi-cab distance because it resembles the route of a car in a grid-like neighborhood such as Manhattan. If a car is travelling from point (0,1) to point (1,0), the shortest route requires a length of 2 units.

- The L_2 norm is the Euclidian length of a vector, $(\sum(x_n)^2)^{1/2}$. It is the most direct route one can possibly take on a geometric plane to get from one point to another. For the mathematically inclined, this is the norm that implements the least square estimation as predicted by the Gauss-Markov theorem. For the rest of you, it's the shortest distance between two points in space.

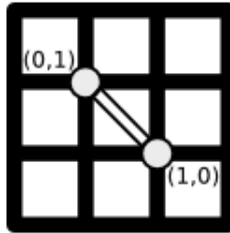


Figure 1.11 The L_2 norm between points (0,1) and (1,0) is the length of a single straight line segment reaching both points.

- The $L-N$ norm generalizes this pattern, resulting in $(\sum|x_n|^N)^{1/N}$. We rarely use finite norms above L_2 , but it's here for completeness.
- The L -infinity norm is $(\sum|x_n|^\infty)^{1/\infty}$. More naturally, it is the largest magnitude among each element. If the vector is (-1, -2, -3), the L -infinity norm will be 3. If a feature vector represents costs of various items, then minimizing the L -infinity norm of the vector is an attempt to reduce the cost of the most expensive item.

When do I use a metric other than the L_2 norm in the real-world?

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

A good goal might be that users shouldn't see five or more incorrect search-results per month. A year's worth of user-data is a 12-dimension vector (where each month of the year is a dimension), indicating number of incorrect results shown per month. You are trying to satisfy the condition that the L -infinity norm of this vector must less than 5.

Suppose instead that your boss changes his/her mind and requires that less than 5 erroneous search-results are allowed for the entire year. In this case, you are trying to achieve a L_1 norm below 5, because the sum of all errors in the entire space should be less than 5.

Actually, your boss changes his or her mind again. Now, the number of months with erroneous search-results should be less than 5. In that case, you are trying to achieve an L_0 norm below 5, because the number of months with a non-zero error should be less than 5.

1.4 Types of Learning

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.

1.4.1 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^* (Theta star) that minimizes the cost among all data points $x \in X$. One way of formalizing this optimization problem is the following:

$$\theta^* = \arg \min_{\theta} \text{Cost}(\theta|X)$$

$$\text{where } \text{Cost}(\theta|X) = \sum_{x \in X} ||g(x|\theta) - f(x)||$$

Clearly, brute forcing every possible combination of θ s, 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, because we already know it works for the training data – we need to tell if it works for data that *wasn't* part of the training set, to make sure our model is “general purpose” and not *biased* to the data we 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?

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.

1.4.2 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 organizing your books onto three shelves, you likely place similar genres together, or maybe you group them by author's last name. You might have a Stephen King section, another for textbooks, and a third for “anything else.” You don't care that they are all

separated by the same feature, just that each has something unique about it that allows you to break it into roughly equal, easily identifiable groups. 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.

1.4.3 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, which might be to solve a maze or accomplish some explicit goal.

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, because we do not know exactly which action in the

history of action sequences to blame for ending up in a low-value state. 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

1.5 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 Apache *Mahout* lower-level library is used to interact with these parallel architectures, 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.

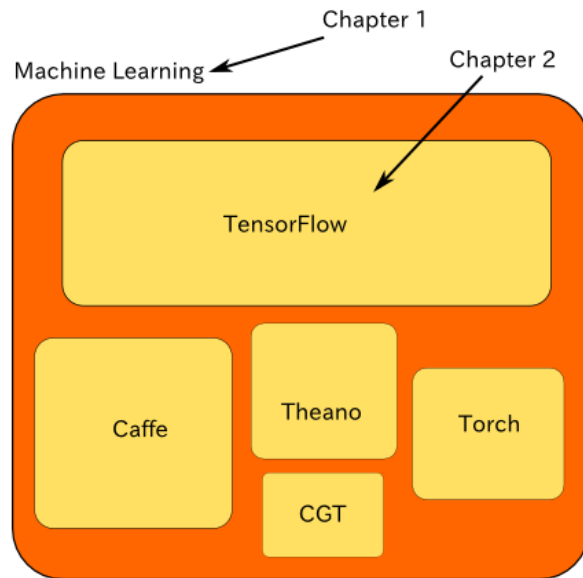


Figure 1.12 TensorFlow is one the most popular machine learning libraries. At the time of writing, it has more than twice as many favorites on GitHub than Caffe, the next most popular library. The size of the rectangles roughly corresponds to popularity on GitHub.

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

1.5.1 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.

1.5.2 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.

1.5.3 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 you wouldn't need to reinvent the wheel.

1.5.4 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.

1.6 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. Because 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.

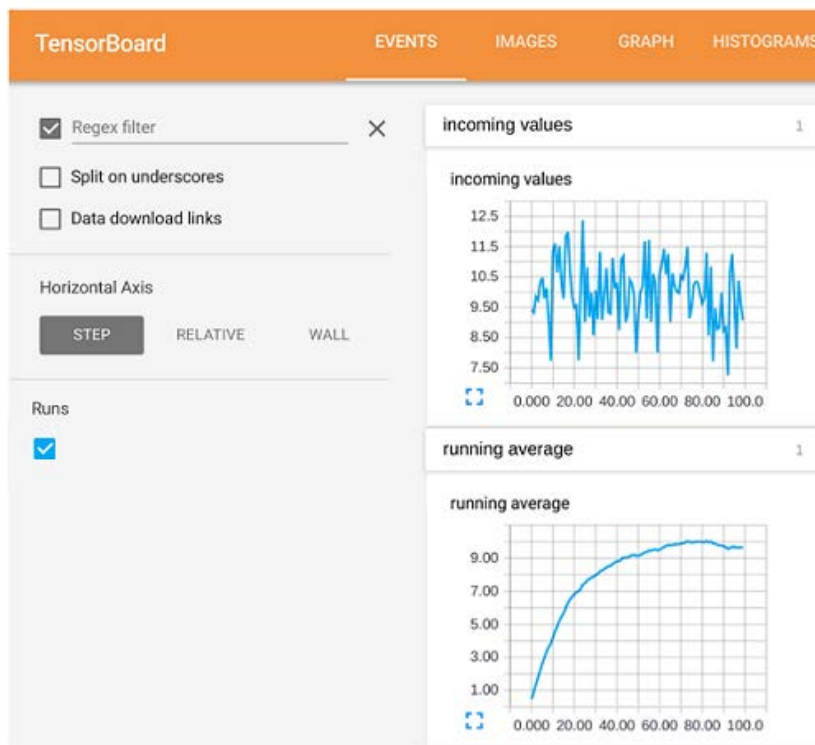


Figure 1.13 Example of TensorBoard in action

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 as well as scalable.

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.

1.7 Overview of future chapters

Chapter 2 demonstrates how to use various components of TensorFlow. 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.

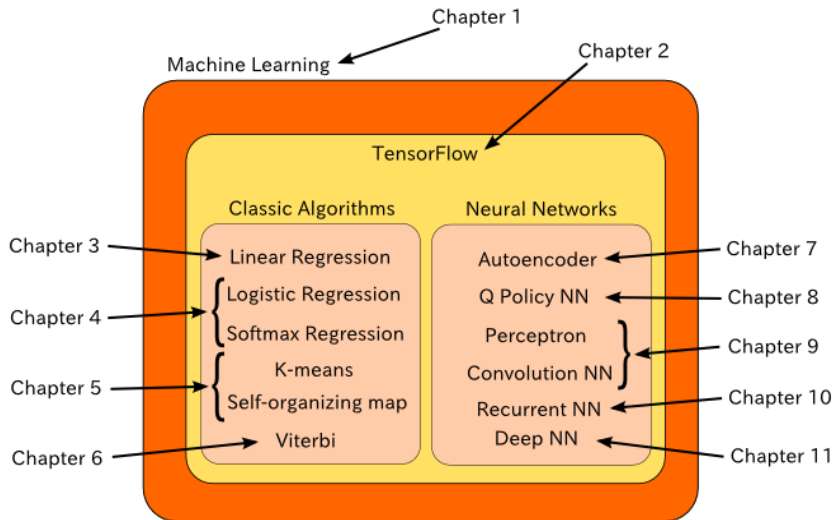


Figure 1.14 The technical chapters are divided into two categories of algorithms: (1) classic algorithms, and (2) neural networks.

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

1.8 Summary

TensorFlow has become the tool of choice among professionals and researchers to implement machine-learning solutions. The next chapter goes into detail about the inner workings of TensorFlow's interface. Chapters 3 through 6 explore how TensorFlow can implement classical algorithms, whereas chapters 7 through 10 demonstrate TensorFlow's prowess in modern neural network architectures.

If you're interested in the intricate architecture details of TensorFlow, the best available source is the official documentation at <https://www.tensorflow.org/extend/architecture>. This book will sprint ahead and use TensorFlow without slowing down into the breadth of low-level performance tuning. For those interested in cloud services, you may consider Google's solution for professional grade scale and speed <https://cloud.google.com/products/machine-learning/>.

You've 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, scholars 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, several 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.

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 auto mode: 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 number 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/api_docs/.

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
```

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).

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

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.

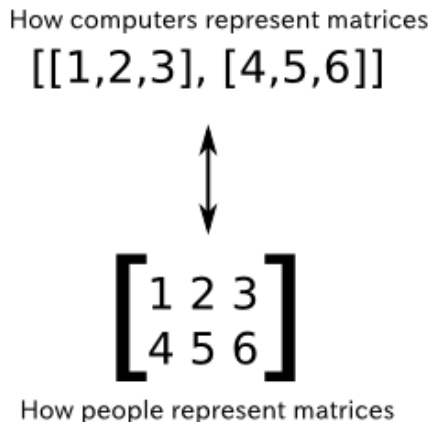


Figure 2.1 The matrix in the lower half of the diagram is a visualization from its compact code notation in the upper half of the diagram. This form of notation is a common paradigm in most scientific computing libraries.

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.

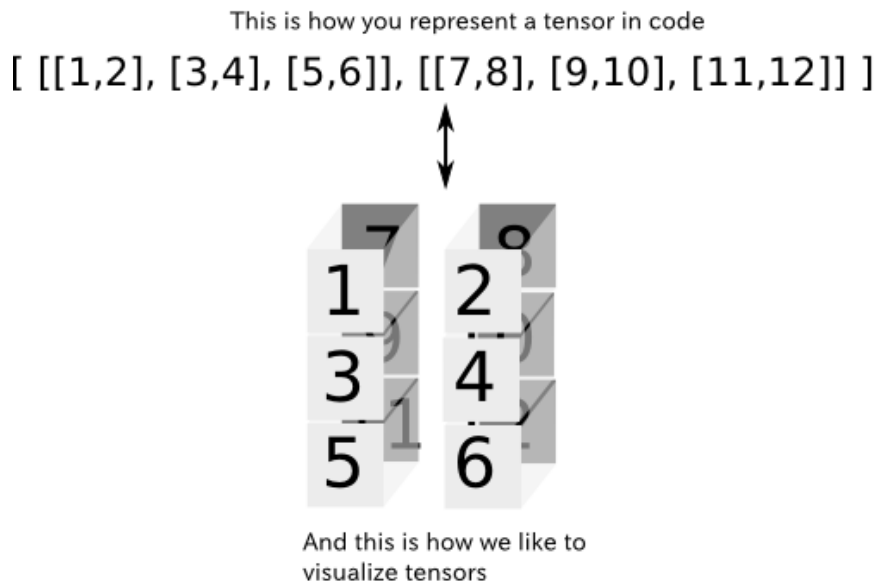


Figure 2.2 This tensor can be thought of as multiple matrices stacked on top of each other. To specify an element, you must indicate the row and column, as well as which matrix is being accessed. Therefore, the rank of this tensor is 3.

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    ##A

m1 = [[1.0, 2.0],
      [3.0, 4.0]]    ##B

m2 = np.array([[1.0, 2.0],
              [3.0, 4.0]], dtype=np.float32)    ##B

m3 = tf.constant([[1.0, 2.0],
                 [3.0, 4.0]])    ##B

print(type(m1))    ##C
print(type(m2))    ##C
print(type(m3))    ##C

t1 = tf.convert_to_tensor(m1, dtype=tf.float32)    ##D
t2 = tf.convert_to_tensor(m2, dtype=tf.float32)    ##D
t3 = tf.convert_to_tensor(m3, dtype=tf.float32)    ##D

print(type(t1))    ##E
print(type(t2))    ##E
print(type(t3))    ##E
```

```
#A We'll use NumPy matrices in TensorFlow
#B Define a 2x2 matrix in 3 different ways
#C Print the type for each matrix
#D Create tensor objects out of the various different types
#E 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 constant `Tensor` object that we initialize using `tf.constant`.

All operators in TensorFlow, such as `negative`, 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'>
```

BY THE WAY You can find these code listings on GitHub, to make copy-and-pasting easier: https://github.com/BinRoot/TensorFlow-Book/blob/master/ch02_basics/Concept01_defining_tensors.ipynb

Let's take another look at defining tensors in code. After importing the TensorFlow library, we can use the `tf.constant` operator as follows in Listing 2.4. Here are a couple different tensors of various dimensions.

Listing 2.4 Creating tensors

```
import tensorflow as tf

m1 = tf.constant([[1., 2.]])    //#A

m2 = tf.constant([[1],
                  [2]])        //#B

m3 = tf.constant([ [1,2],
                   [3,4],
                   [5,6]],
                  [[7,8],
                   [9,10],
                   [11,12]] ])  //#C

print(m1)    //#D
print(m2)    //#D
print(m3)    //#D
```

```
#A Define a 2x1 matrix
#B Define a 1x2 matrix
#C Define a rank 3 tensor
#D 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 `m1` 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 once. 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 a negation op (short for operation) on our `m1` 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]])    ##A
neg_x = tf.negative(x)     ##B
print(neg_x)              ##B
```

```
##A Define an arbitrary tensor
##B Negate the tensor
##C Print the object
```

Listing 2.5 generates the following output:

```
Tensor("Neg:0", shape=TensorShape([Dimension(1), Dimension(2)]), dtype=int32)
```

Notice how the output is not `[[-1, -2]]`. That's because we're printing out the definition of the negation op, not the actual evaluation of the op. The printed output shows that our negation op is a `Tensor` class with a name, shape, and data-type. The name was automatically assigned, but you could've provided it explicitly as well when using the `tf.negative` op in listing 2.5. Similarly, the shape and data-type were inferred from the `[[1, 2]]` that we passed in.

Useful TensorFlow operators

The official documentation carefully lays out all available math ops:

https://www.tensorflow.org/api_guides/python/math_ops.

Some specific examples of commonly used operators include:

`tf.add(x, y)` → Add two tensors of the same type, $x + y$

`tf.subtract(x, y)` → Subtract tensors of the same type, $x - y$

`tf.multiply(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 `**`, `-`, `+`, and so on 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 = 0.0
sigma = 1.0
(tf.exp(tf.negative(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

x = tf.constant([[1., 2.]])    //#A
neg_op = tf.negative(x)      //#B

with tf.Session() as sess:    //#C
    result = sess.run(neg_op)  //#D

print(result)                //#E
```

```
#A Define an arbitrary matrix
#B Run the negation operator on it
#C Start a session to be able to run operations
#D Tell the session to evaluate negMatrix
#E 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. Not only does a session configure *where* your code will be computed on your machine, but it also crafts *how* the computation will be laid out in order to parallelize computation.

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 used `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()    //#A

x = tf.constant([[1., 2.]])    //#B
neg_x = tf.negative(x)    //#B

result = neg_x.eval()    //#C
print(result)    //#D

sess.close()    //#E
```

#A Start an interactive session so the sess variable no longer needs to be passed around

#B Define some arbitrary matrix and negate it

#C You can now evaluate negMatrix without explicitly specifying a session

#D Print the negated matrix

#E Remember to close the session to free up resources

2.4.1 Understanding code as a graph

Consider a doctor who predicts 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. So, whenever you see a plus symbol “+” or any mathematical concept, just picture it as one of many nodes. The edges between these nodes represent the composition of mathematical functions. Specifically, the `negative` operator we’ve been studying so far is a node, and the incoming/outgoing edges of this node are how the Tensor transforms. A tensor *flows* through the graph, which is why this library is called TensorFlow!

Here’s a thought: every operator is a strongly typed function that takes input tensors of a dimension and produces output of the same 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. This graph, in whole, represents a complicated mathematical function (specifically, the Gaussian function). Small segments of the graph represent simple mathematical concepts, such as negation or doubling.

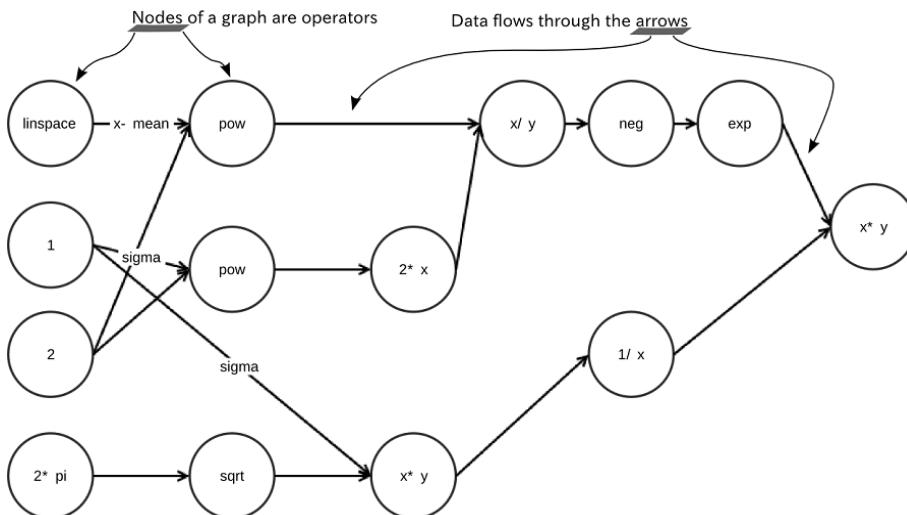


Figure 2.3 The graph represents the operations needed to produce a Gaussian distribution. The links between the nodes represent how data flows from one operation to the next. The operations themselves are very simple, but the complexity arises from how they intertwine.

TensorFlow algorithms are easy to visualize. They can be simply described by flowcharts. The technical (and more correct) term for such a flowchart is a *dataflow graph*. Every arrow in a dataflow graph is called the *edge*. In addition, every state of the dataflow graph is called a *node*. The purpose of the session is to interpret your Python code into a dataflow graph, and then associate the computation of each node of the graph to the CPU or GPU.

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, which will show you exactly where on your hardware the computations are evoked.

Listing 2.8 Logging a session

```
import tensorflow as tf

x = tf.constant([[1., 2.]])  ##A
neg_x = tf.negative(x)  ##A

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:  ##B
    result = sess.run(neg_x)  #C

print(result)  #D
```

```
#A Define a matrix and negate it
#B Start the session with a special config passed into the constructor to enable logging
#C Evaluate negMatrix
#D 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. Typically, placeholders are the input and output of your model.
- Variable

A value that can change, such as parameters of a machine learning model. Variables must be initialized by the session before they are used.
- Constant

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

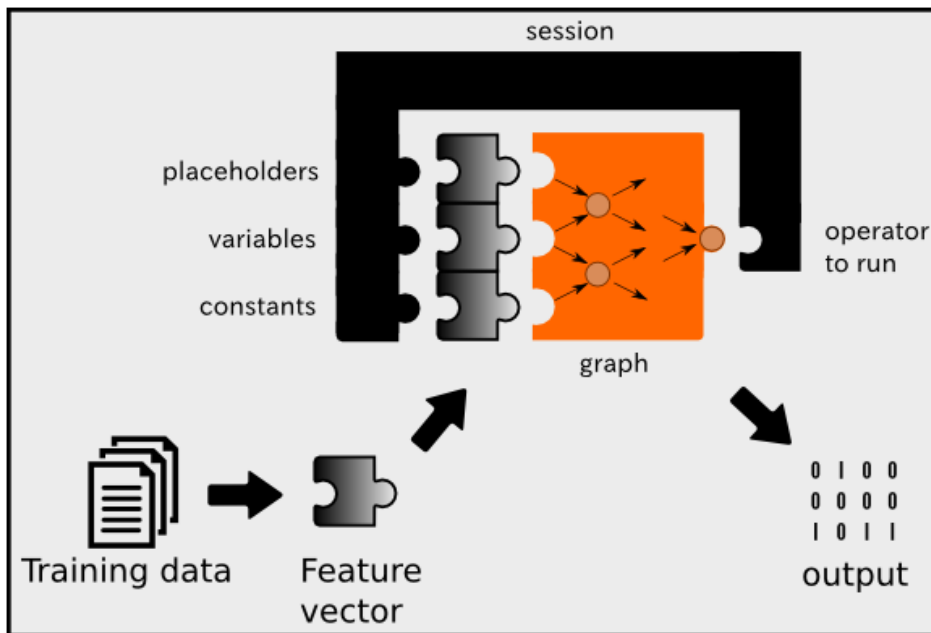


Figure 2.4 The session dictates how the hardware will be used to process the graph most efficiently. When the session starts, it assigns the CPU and GPU devices to each of the nodes. After processing, the session outputs data in a usable format, such as a NumPy array. A session optionally may be fed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

The entire machine learning with TensorFlow pipeline follows the flow of figure 2.4. Most of the code in TensorFlow is about setting up the graph and session. Once you design a graph and hook up the session to execute it, your code is ready to use!

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 others 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 3". 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. They 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 are three common ways to evaluate cells:

1. Pressing `Shift+Enter` on a cell executes the cell and highlights the next cell below.
2. Pressing `Ctrl+Enter` will maintain the cursor on the current cell after executing it.
3. 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.

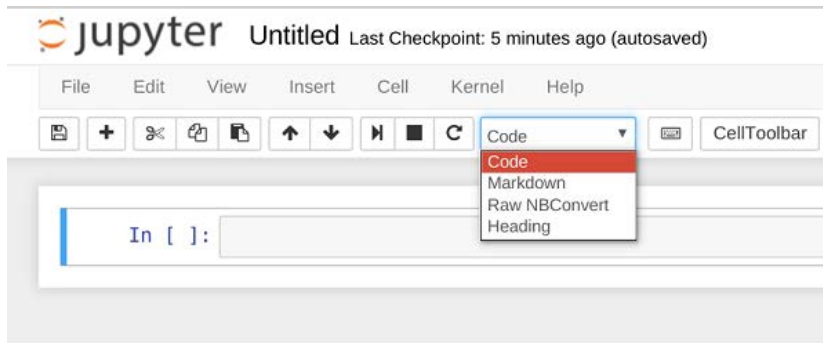


Figure 2.6 The dropdown menu changes the type of cell in the notebook. The “Code” cell is for Python code, whereas the “Markdown” code is for text descriptions.

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.

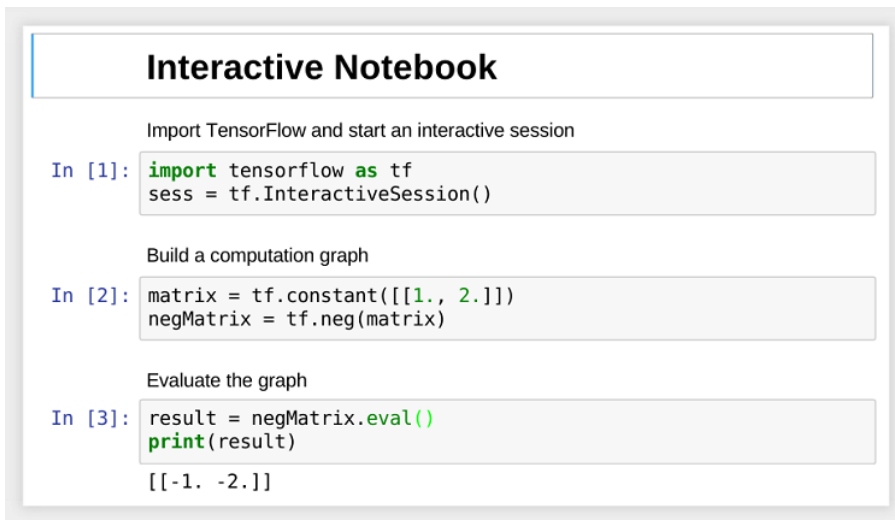


Figure 2.7 An interactive Python notebook presents both code and comments side by side.

EXERCISE 2.3 If you look closely at Figure 2.7, you’ll notice that it uses `tf.neg` instead of `tf.negative`. That’s strange. Could you explain why I might have done that?

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. Programs with only constants aren't that interesting for real-world applications, so TensorFlow allows richer tools such as variables, which 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 for each variable. In the world of machine learning, it is common for parameters to fluctuate until eventually settling down, making variables an excellent 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. TensorFlow allows us to declare a session using `tf.InteractiveSession()`. When you've declared an interactive session, TensorFlow functions don't require the session attribute they would otherwise, which makes coding in Jupyter notebooks easier.

Listing 2.9 Using a variable

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

raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]    ##B
spike = tf.Variable(False)    ##C
spike.initializer.run()    ##D

for i in range(1, len(raw_data)):    ##E
    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()    ##G
```

#A Start the session in interactive mode so we won't need to pass around sess

```

#B Let's say we have some raw data like this
#C Create a Boolean variable called spike to detect a sudden increase in a series of numbers
#D Because all variables must be initialized, initialize the variable by calling run() on its initializer
#E Loop through the data (skipping the first element) and update the spike variable when there is a significant increase
#F To update a variable, assign it a new value using tf.assign(<var name>, <new value>). Evaluate it to see the change.
#G 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 with the same name. Naming a variable is optional, but highly encouraged to organize your code.

Try running this code to see the results.

Listing 2.10 Saving variables

```

import tensorflow as tf  ##A
sess = tf.InteractiveSession()  ##A

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

saver = tf.train.Saver()  ##E

for i in range(1, len(raw_data)):  ##F
    if raw_data[i] - raw_data[i-1] > 5:  ##F
        spikes_val = spikes.eval()  ##G
        spikes_val[i] = True  ##G
        updater = tf.assign(spikes, spikes_val)  ##G
        updater.eval()  ##H

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

```

```
sess.close()
```

```
#A Import TensorFlow and enable interactive sessions
#B Let's say we have a series of data like this
#C Define a boolean vector called spike to locate a sudden spike in raw data
#D Don't forget to initialize the variable
#E 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.
#F Loop through the data and update the spike variable when there is a significant increase
#G Update the value of spikes by using the tf.assign function
#H Don't forget to actually evaluate the updater, otherwise spikes will not be updated
#I Save the variable to disk
# Print out where the relative file path of the saved variables
```

You will notice a couple files generated, one of them being `spikes.ckpt`, 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') // #A
# spikes.initializer.run() // #B
saver = tf.train.Saver() // #C

saver.restore(sess, "./spikes.ckpt") // #D
print(spikes.eval()) // #E

sess.close()
```

```
#A Create a variable of the same size and name as the saved data
#B You no longer need to initialize this variable because it will be directly loaded
#C Create the saver op to restore saved data
#D Restore data from the "spikes.ckpt" file
#E 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.

Let's see how we can visualize variable trends over time in a real-world example. In the next section, we'll implement a moving-average algorithm in TensorFlow and then in the section after, we'll carefully track the variables we care about for visualization in TensorBoard.

2.8.1 Implementing a moving average

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, $Mean = (x_1 + x_2 + \dots + x_n) / n$. 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 points. The exponential average algorithm calculates the current estimated average as a function of the previous estimated average and the current value.

More succinctly, $Avg_t = f(Avg_{t-1}, x_t) = (1 - a) Avg_{t-1} + a x_t$. Alpha (a) 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 a , 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 $Avg_t = (1 - a) Avg_{t-1} + a x_t$. 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 //#A
```

#A alpha is a tf.constant, curr_value is a placeholder, and prev_avg is a variable

We'll define the undefined variables later. The reason we're writing code in such backward way is because by defining the interface first, it forces us to implement the peripheral setup code to satisfy the interface. 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)  // #A

alpha = tf.constant(0.05)  // #B
curr_value = tf.placeholder(tf.float32)  // #C
prev_avg = tf.Variable(0.)  // #D
update_avg = alpha * curr_value + (1 - alpha) * prev_avg

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for i in range(len(raw_data)):  // #E
        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)
```

#A Create a vector of 100 numbers with a mean of 10 and standard deviation of 1

#B Define alpha as a constant

#C A placeholder is just like a variable, but the value is injected from the session

#D Initialize the previous average to zero

#E Loop through the data one-by-one to update the average

2.8.2 Visualizing the moving average

Now that we have a working implementation of a moving average algorithm, let's visualize the results using TensorBoard. Visualization using TensorBoard is usually a two-step process.

1. First, you must pick out which nodes you really care about measuring by annotating them with a *summary op*.
2. Then, call `add_summary` on them to queue up data to be written to disk

For example, let's say we have an *img* placeholder and a *cost* op, as shown in listing 2.15. You can annotate each of them (by giving them a name such as "img" or "cost") so that they are capable of being visualized in TensorBoard. We're going to do something similar with our moving-average example.

Listing 2.15 Annotating with Summary op

```
img = tf.placeholder(tf.float32, [None, None, None, 3])
cost = tf.reduce_sum(...)

my_img_summary = tf.summary.image("img", img)
```



```
my_cost_summary = tf.summary.scalar("cost", cost)
```

More generally, 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.

WARNING Be careful not to call the `add_summary` function too often! Although doing so will produce higher resolution visualizations of your variables, it'll be at the cost of more computation and slightly slower learning.

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 2.16 shows you how to hook up the `SummaryWriter` to your code. Run it and refresh the TensorBoard to see the visualizations.

Listing 2.16 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.summary.scalar("running_average", update_avg) // #A
value_hist = tf.summary.scalar("incoming_values", curr_value) // #B
merged = tf.summary.merge_all() // #C
writer = tf.summary.FileWriter("./logs") // #D
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    sess.add_graph(sess.graph) // #E
    for i in range(len(raw_data)):
        summary_str, curr_avg = sess.run([merged, update_avg], feed_dict={curr_value:
raw_data[i]}) // #F
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
        writer.add_summary(summary_str, i) // #G
```

- #A Create a summary node for the averages
- #B Create a summary node for the values
- #C Merge the summaries to make it easier to run together
- #D Pass in the “logs” directory location to the writer
- #E Optional, but it allows you to visualize the computation graph in TensorBoard
- #F Run the merged op and the update_avg op at the same time
- #G Add the summary to the writer

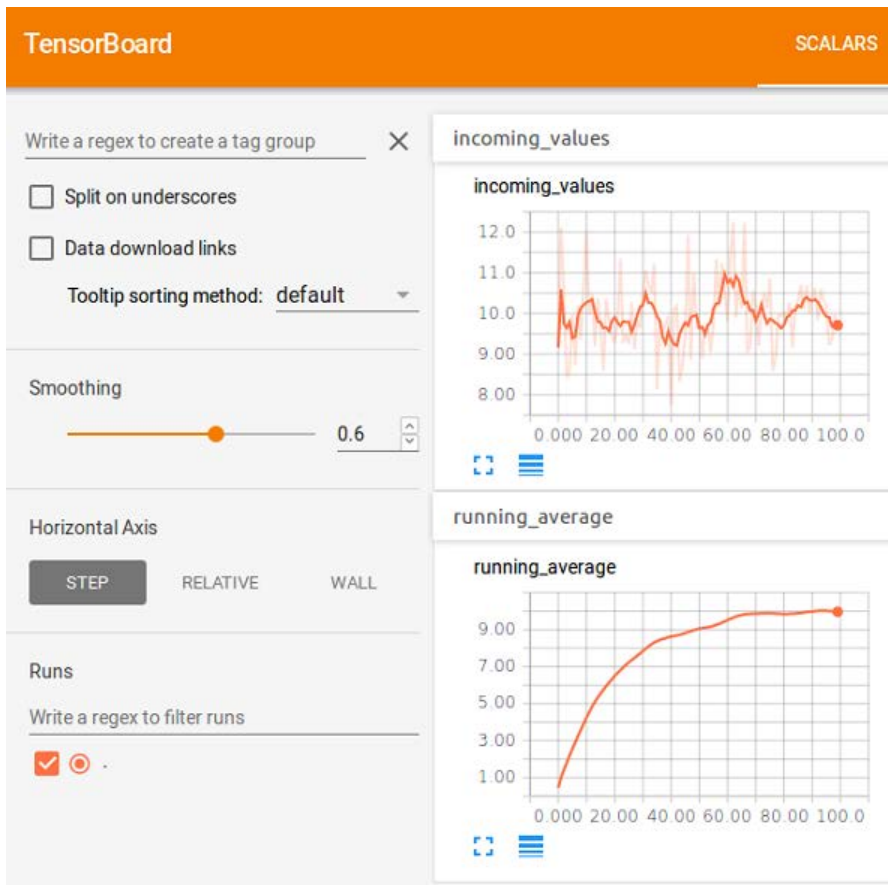


Figure 2.8 TensorBoard provides a user-friendly interface to visualize data produced in TensorFlow.

BY THE WAY You may need to ensure the TensorFlow session has ended before starting TensorBoard. If you rerun listing 2.16, you'll need to remember to clear the logs directory.

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.

1. 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.
2. 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).
3. TensorBoard provides an easy way to visualize how data changes in TensorFlow code as well as troubleshoot bugs by inspecting trends in data.
4. 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 affects 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 might 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."

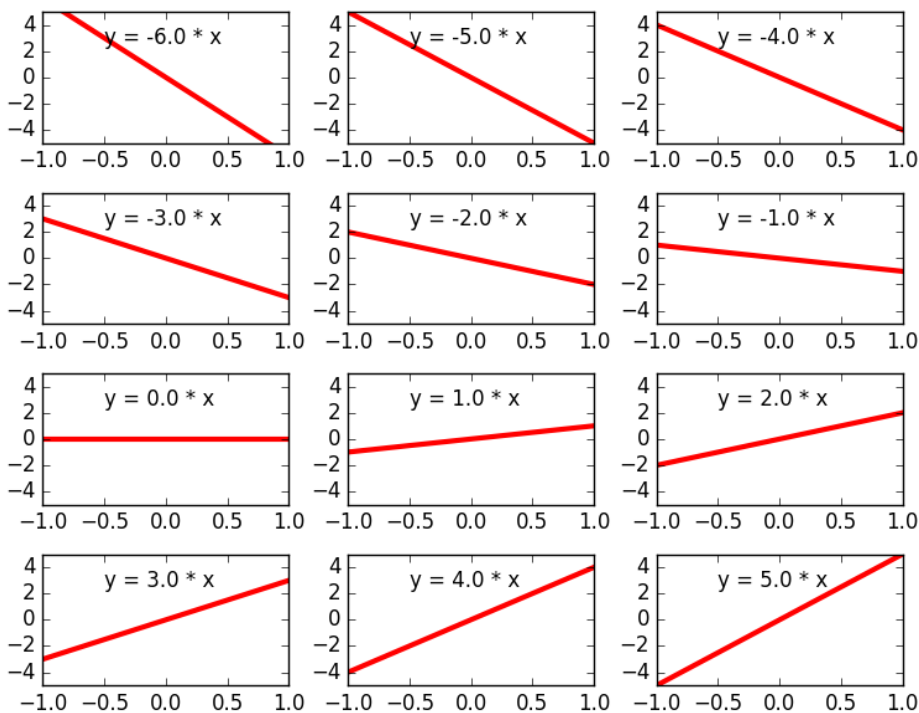


Figure 3.1 Different values of the parameter w result in different linear equations. The set of all these linear equations is what constitutes the linear model M .

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} .

DID YOU KNOW? Regression can also be posed with multiple outputs, as opposed to just one real number. In that case, we call it *Multivariate Regression*.

The input of the function could be continuous or discrete. However, the output must be continuous, as demonstrated by figure 3.2.

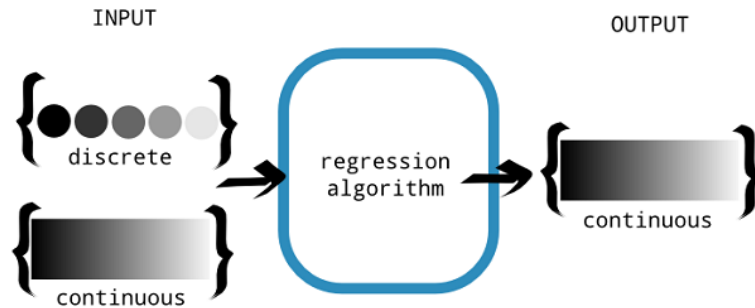


Figure 3.2 A regression algorithm is meant to produce continuous output. The input is allowed to be discrete or continuous. This distinction is important because discrete-valued outputs are handled better by classification, which is discussed in the next chapter.

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 example, $f(0) = 0$, $f(1) = 1$, 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).

Train	Test	Result
👍	👍	👍 Ideal
👎	👎	👎 Underfit
👍	👎	👎 Overfit

Figure 3.3 Ideally, the best fit curve fits well on both the training data as well as the test data. If we witness it fitting poorly with the test data and the training data, there's a chance that our model is underfitting. On the other hand, if it performs poorly on the test data but well on the training data, then we know the model is overfitting.

On the other end of the spectrum, a not-so-flexible model may generalize better to unseen testing data, but would score relatively low on 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 of a model underfitting and overfitting data-points in 2D.

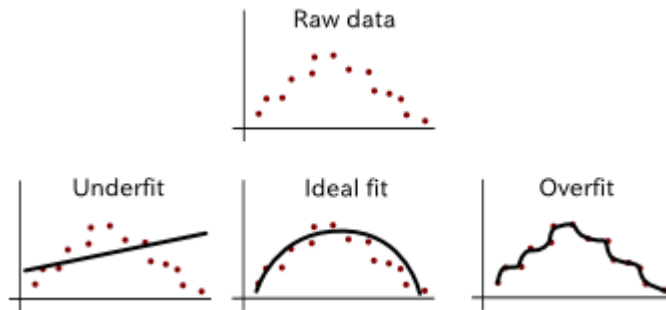


Figure 3.4 Examples of under-fitting and over-fitting the data.

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, $\{y=0, y=x, y=2x, \dots, y=9x\}$.

In summary, measuring how well your model does on the training data is not a great indicator of its generalizability. Instead, you should really evaluate your model on a separate batch of testing data. You might find out that it performs terribly on the test data, in which case your model is likely overfitting to the training data. If the testing error is actually around the same as the train error, and both errors are similar, then your model is probably underfitting.

Which is why 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  // #A
```

```
import matplotlib.pyplot as plt  ##B
x_train = np.linspace(-1, 1, 101)  ##C
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33  ##D

plt.scatter(x_train, y_train)  ##E
plt.show()  ##E
```

#A Import NumPy to help generate initial raw data

#B Use matplotlib to visualize the data

#C The input values are 101 evenly spaced numbers between -1 and 1

#D The output values are proportional to the input but with added noise

#E Use matplotlib's function to generate a scatter plot of the data

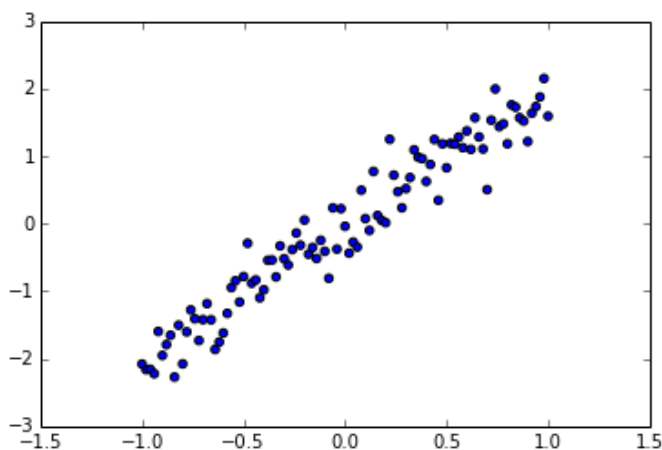


Figure 3.5 Scatter plot of $y = x + \text{noise}$.

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 looping through all your data to update the parameters is called an *epoch*.

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

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.

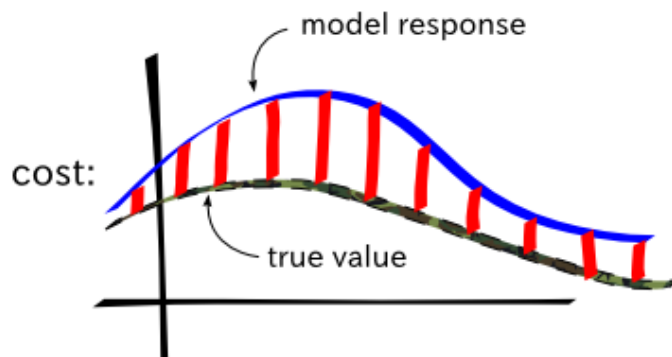


Figure 3.7 The cost is the norm of the point-wise difference between the model response and the true value.

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  ##A
import numpy as np  ##A
import matplotlib.pyplot as plt  ##A

learning_rate = 0.01  ##B
training_epochs = 100  ##B

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

```

X = tf.placeholder(tf.float32) //#D
Y = tf.placeholder(tf.float32) //#D

def model(X, w): //#E
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights") //#F

y_model = model(X, w) //#G
cost = tf.square(Y-y_model) //#G

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) //#H

sess = tf.Session() //#I
init = tf.global_variables_initializer() //#I
sess.run(init) //#I

for epoch in range(training_epochs): //#J
    for (x, y) in zip(x_train, y_train): //#K
        sess.run(train_op, feed_dict={X: x, Y: y}) //#L

w_val = sess.run(w) //#M

sess.close() //#N
plt.scatter(x_train, y_train) //#O
y_learned = x_train*w_val //#P
plt.plot(x_train, y_learned, 'r') //#P
plt.show() //#P

```

#A 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.

#B Define some constants used by the learning algorithm. There are called hyper-parameters.

#C Set up fake data that we will use to find a best fit line

#D Set up the input and output nodes as placeholders since the value will be injected by `x_train` and `y_train`.

#E Define the model as $y = w * x$

#F Set up the weights variable

#G Define the cost function

#H Define the operation that will be called on each iteration of the learning algorithm

#I Set up a session and initialize all variables

#J Loop through the dataset multiple times

#K Loop through each item in the dataset

#L Update the model parameter(s) to try to minimize the cost function

#M Obtain the final parameter value

#N Close the session

#O Plot the original data

#P Plot the best fit line

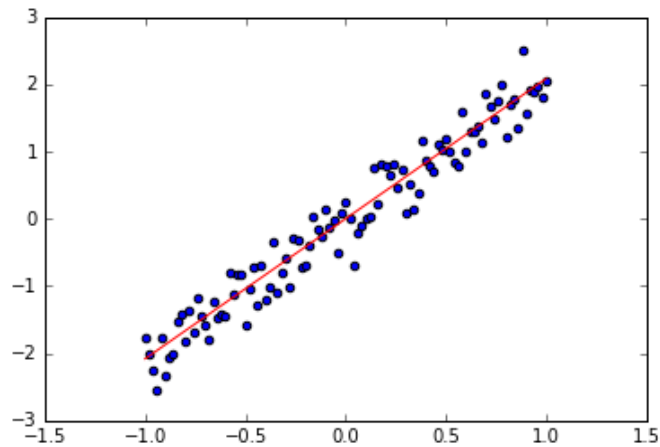


Figure 3.8 Linear regression estimate shown by running listing 3.2.

As shown in figure 3.8 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 3.9.

Learning algorithm in TensorFlow

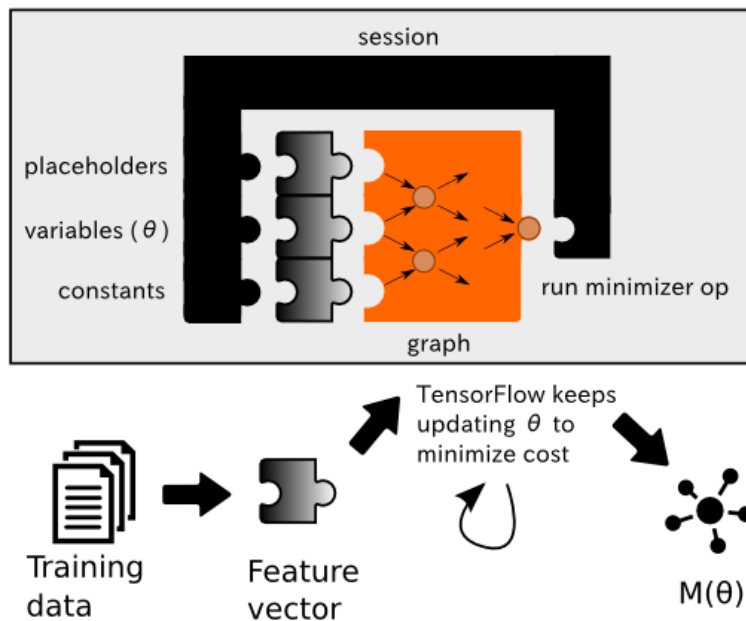


Figure 3.9 The learning algorithm updates the model's parameters to minimize the given cost function.

We've just learned how to implement a simple regression model in TensorFlow. Making further improvements is simply a matter of enhancing the model with the right medley of variance and bias, as we discussed earlier. For example, the linear regression model we've designed so far is burdened with a strong bias, meaning it only expresses a limited set of function, such as linear functions. In the next section, we'll try a more flexible model. You'll notice how only the TensorFlow graph needs to be rewired, while everything else (such as preprocessing, training, evaluation) stays the same.

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:

$$f(x) = w_n x^n + \dots + w_1 x + w_0$$

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

Consider the scatter plot in figure 3.10, 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.

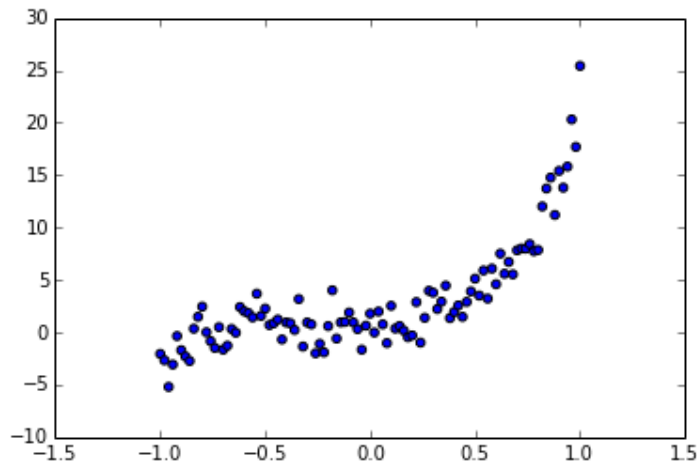


Figure 3.10 Data points like this are not suitable for a linear model.

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 ##A
import numpy as np ##A
import matplotlib.pyplot as plt ##A

learning_rate = 0.01 ##A
training_epochs = 40 ##A

trX = np.linspace(-1, 1, 101) ##B

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

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

plt.scatter(trX, trY) ##E
plt.show() ##E

X = tf.placeholder(tf.float32) ##F
Y = tf.placeholder(tf.float32) ##F

def model(X, w): ##G
    terms = [] ##G
    for i in range(num_coeffs): ##G
        term = tf.multiply(w[i], tf.pow(X, i)) ##G
```

```

        terms.append(term)    ##G
    return tf.add_n(terms)    ##G

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

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

sess = tf.Session()    ##J
init = tf.global_variables_initializer()    ##J
sess.run(init)    ##J

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

w_val = sess.run(w)    ##J
print(w_val)    ##J

sess.close()    ##K

plt.scatter(trX, trY)    ##L
trY2 = 0    ##L
for i in range(num_coeffs):    ##L
    trY2 += w_val[i] * np.power(trX, i)    ##L

plt.plot(trX, trY2, 'r')    ##L
plt.show()    ##L

```

```

#A Import the relevant libraries and initialize the hyper-parameters
#B Set up some fake raw input data
#C Set up raw output data based on a degree 5 polynomial
#D Add some noise
#E Show a scatter plot of the raw data
#F Define the nodes to hold values for input/output pairs
#G Define our polynomial model
#H Set up the parameter vector to all zeros
#I Define the cost function just as before
#J Set up the session and run the learning algorithm just as before
#K Close the session when done
#L Plot the result

```

The final output of listing 3.3 is a 5th-degree polynomial that fits the data, as seen in figure 3.11.

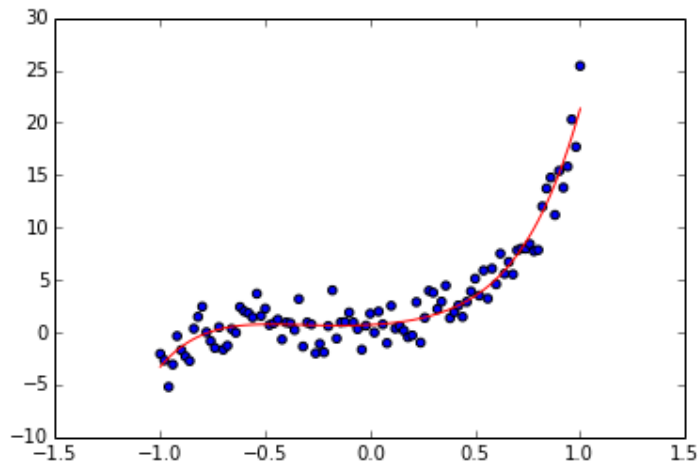


Figure 3.11 The best fit curve smoothly aligns with the nonlinear data

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 3.12. We try to fit an 8th 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.

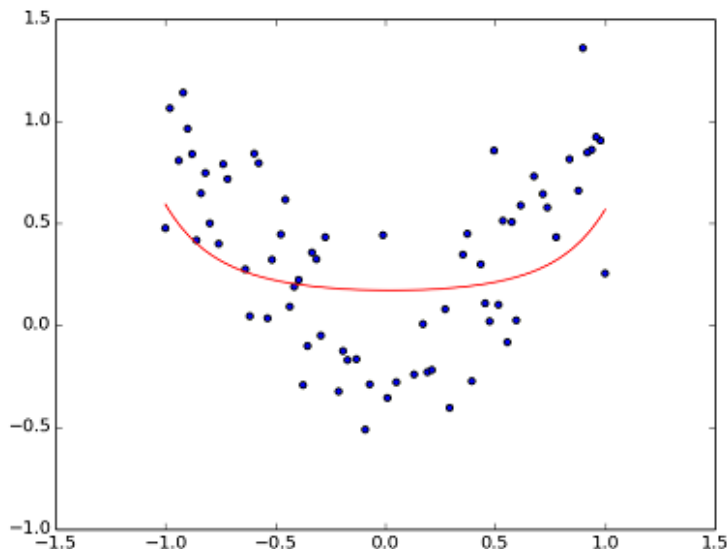


Figure 3.12 When the model is too flexible, a best fit curve could look awkwardly complicated or unintuitive. We need to use regularization to improve the fit, so that the learned model performs well against test data.

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 2nd 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): //#A
    arr = np.arange(x_dataset.size) //#B
    np.random.shuffle(arr) //#B
    num_train = int(ratio * x_dataset.size) //#C
    x_train = x_dataset[arr[0:num_train]] //#D
    y_train = y_dataset[arr[0:num_train]] //#D
    x_test = x_dataset[arr[num_train:x_dataset.size]] //#E
    y_test = y_dataset[arr[num_train:x_dataset.size]] //#E
    return x_train, x_test, y_train, y_test //#F
```

#A Take the input and output dataset as well as the desired split ratio

#B Shuffle a list of numbers

#C Calculate the number of training examples

#D Use the shuffled list to split the x_dataset

#E Likewise, split the y_dataset

#F Return the split x and y datasets

EXERCISE 3.3 There is a Python library called scikit-learn that supports many useful data preprocessing algorithms. In fact, we can call a function in scikit-learn to do exactly what listing 3.4 achieves. Can you find this function on the library's documentation? Hint: http://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

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 //#A
import numpy as np //#A
import matplotlib.pyplot as plt //#A

learning_rate = 0.001 //#A
training_epochs = 1000 //#A
reg_lambda = 0. //#A

x_dataset = np.linspace(-1, 1, 100) //#B

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

(x_train, x_test, y_train, y_test) = split_dataset(x_dataset, y_dataset, 0.7) //#C

X = tf.placeholder(tf.float32) //#D
Y = tf.placeholder(tf.float32) //#D

def model(X, w): //#E
    terms = [] //#E
```

```

for i in range(num_coeffs): //#E
    term = tf.multiply(w[i], tf.pow(X, i)) //#E
    terms.append(term) //#E
return tf.add_n(terms) //#E

w = tf.Variable([0.] * num_coeffs, name="parameters") //#F
y_model = model(X, w) //#F
cost = tf.div(tf.add(tf.reduce_sum(tf.square(Y-y_model)),
                    tf.multiply(reg_lambda, tf.reduce_sum(tf.square(w)))),
              2*x_train.size) //#F
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) //#F

sess = tf.Session() //#G
init = tf.global_variables_initializer() //#G
sess.run(init) //#G

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

sess.close() //#I

```

```

#A Import the relevant libraries and initialize the hyper-parameters
#B Create a fake dataset.  $y = x^2$ 
#C Split the dataset into 70% training and testing 30% using listing 3.4
#D Set up the input/output placeholders
#E Define our model
#F Define the regularized cost function
#G Set up the session
#H Try out various regularization parameters
#I 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.

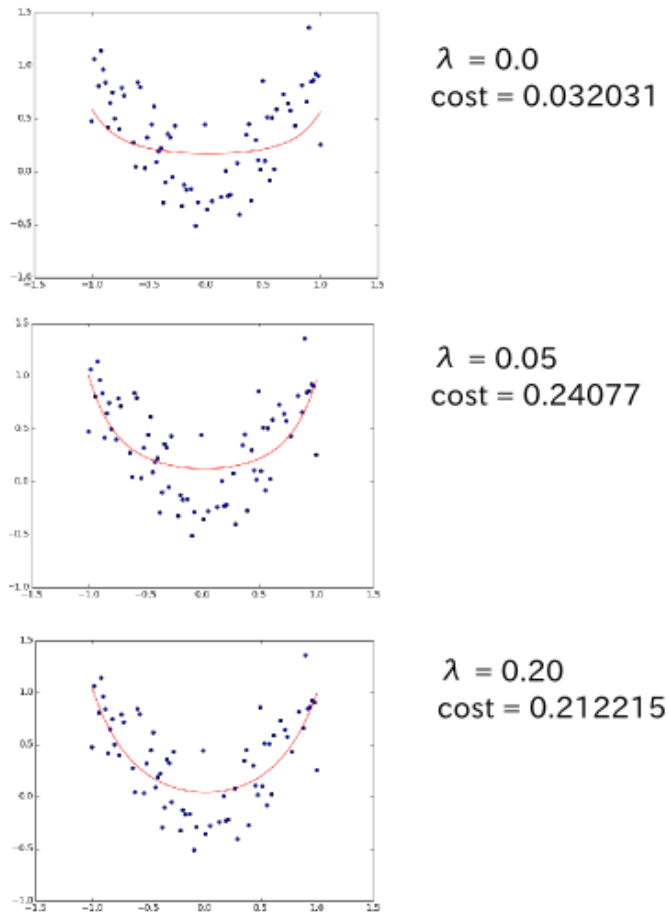


Figure 3.13 As we increase the regularization parameter to some extent, the cost decreases. This implies that the model was originally overfitting the data, and regularization helped add some structure.

3.5 Application of linear regression

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.

1. University of Massachusetts Amherst supplies small datasets of various types.
 - o <http://www.umass.edu/statdata/statdata/>
2. Kaggle contains all types of large scale data for machine learning competitions.
 - o <https://www.kaggle.com/datasets>

3. Data.gov is an open data initiative by the US government, which contains many interesting and practical datasets.

- o <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    ##A
import time  ##B

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

    days_in_year = 365

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

    ##D
    with open(filename, 'rb') as csvfile:
        csvreader = csv.reader(csvfile)
        csvreader.next()
        for row in csvreader:
            if row[date_idx] == '':
                continue
            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)  ##E
```

#A For easily reading csv files

#B For using useful date functions

#C Set up initial frequency map

#D Read data and aggregate count per period

#E Obtain a weekly frequency count of 3-1-1 phone calls in 2014

The code in listing 3.6 gives you the training data for linear regression. The `freq` variable is a dictionary that maps a period (such as a week) to a frequency count. A year has 52 weeks, so you'll have 52 data points, if we leave `bucket=7` as is.

Now that you have data points, you have exactly the input and output necessary to fit a regression model using the techniques covered in this chapter. More practically, the learned model can be used to interpolate or extrapolate frequency counts.

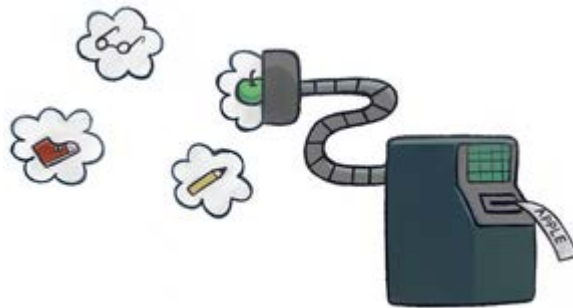
3.6 Summary

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.

1. Regression is a type of supervised machine learning for predicting continuous-valued output.
2. 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.
3. We can easily modify linear regression to learn polynomials or other more complicated curves.
4. To avoid overfitting our data, we regularize the cost function by penalizing larger valued parameters.
5. If the output of the function is not continuous, a classification algorithm should instead be used (see the next Chapter).
6. 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 Easy to update model with new data	Only supports binary labels
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*.

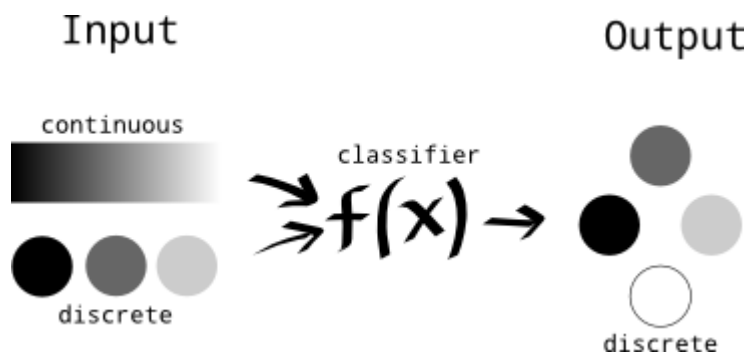


Figure 4.1 A classifier produces discrete outputs, but may take either continuous or discrete inputs.

Formally, a category label is restricted to a range of possible values. You can think of two-valued labels like Boolean variables in Python. When the input features only have a fixed set of possible values, we need to ensure our model can understand how to handle them. Since the set of functions in a model typically deal with continuous real numbers, we'll need to preprocess the dataset to account for discrete variables, which fall into one of two types: ordinal or nominal (figure 4.2).

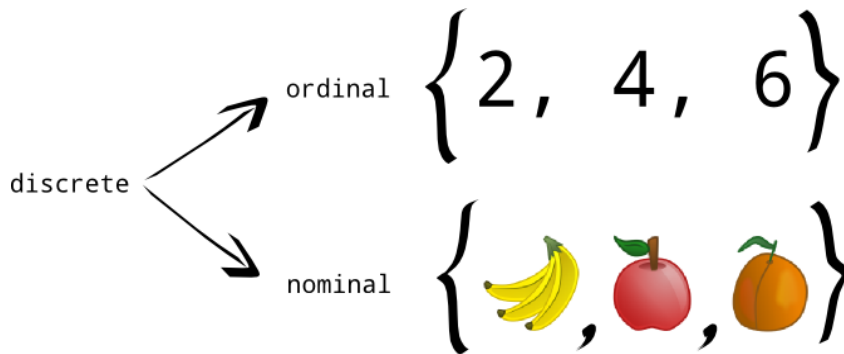


Figure 4.2 There are two types of discrete sets, those that can be ordered (*ordinal*) and those that cannot (*nominal*).

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 $\{\textit{banana}, \textit{apple}, \textit{orange}\}$ 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 $\{\textit{banana}, \textit{apple}, \textit{orange}\}$ 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. This is often referred to as *one-hot encoding*.

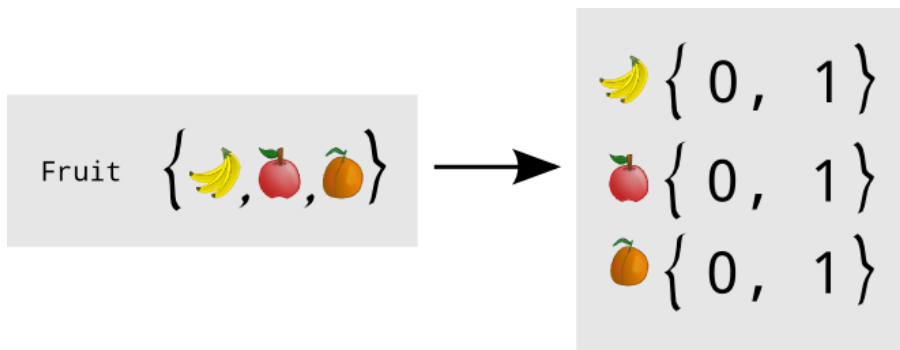


Figure 4.3 If the values of a variable are discrete, then they might need to be preprocessed. One solution is to treat each of the discrete values as a Boolean variable, as shown on the right. Banana, apple, and orange are 3 newly added variables, each having values 0 or 1. The original “fruit” variable 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 which stocks you should buy, sell, or hold, (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 grade-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:

$$accuracy = \frac{\#correct}{\#total}$$

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). As we can see, it’s called a confusion matrix because it makes it easy to see how often a model confuses two classes that it’s trying to differentiate.

		Predicted	
		Positive (Green Check)	Negative (Red Cross)
Actual	Positive (Green Check)	TP	FN
	Negative (Red Cross)	FP	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:

$$precision = \frac{TP}{TP + FP}$$

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:

$$recall = \frac{TP}{TP + FN}$$

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

Let's do a quick example. Let's say we are trying to identify cats in a set of 100 pictures. 40 of the pictures are cats, and 60 are dogs. When we ran our classifier, 10 of the cats were identified as dogs, and 20 of the dogs were identified as cats. Our confusion matrix looks like this:

confusion matrix

		Predicted	
		Cat	Dog
Actual	Cat	30 True positives	20 False Positives
	Dog	10 False Negatives	40 True Negatives

You can see the total number of cats on the left side – 30 identified correctly, and 10 not, totaling 40.

EXERCISE 4.X What's the precision and recall for cats? What's the accuracy of the system?

ANSWER For cats, our precision is $30/(30+20)$ or $3/5$. The recall is $30/(30+10)$, or $3/4$. The accuracy is $(30+40)/100$, or 70%.

4.2.3 Receiver operating characteristic curve

Because binary classifiers are among the most popular tools, many mature techniques exist for measuring their performance, such as the receiver operating characteristic (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. Good algorithms are way above the baseline. A quantitative way to compare classifiers is by measuring the area under the ROC curve. If a model has an area-under-curve (AUC) value higher than 0.9, then it's an excellent classifier. A model that randomly guesses the output will have an AUC value of about 0.5. See figure 4.5 for an example.

ROC Curves

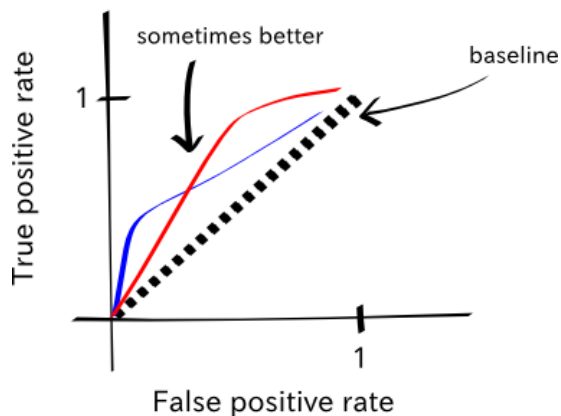
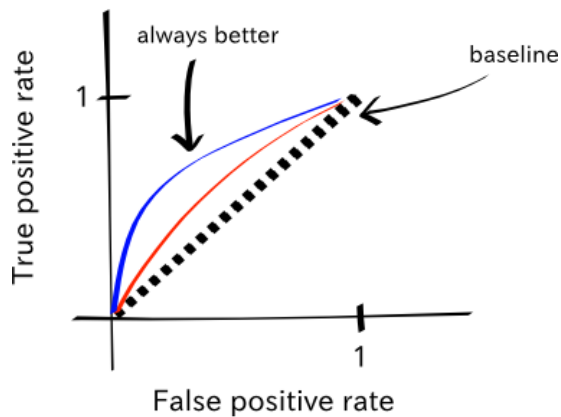


Figure 4.5 The principled way to compare different algorithms is by examining their ROC curves. When the true-positive rate is greater than the false-positive rate at every situation, then it's straightforward to declare that one algorithm is dominant in terms of its performance. If the true-positive rate is less than the false-positive rate, the plot dips below the baseline shown by a dotted-line.

EXERCISE 4.2 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.

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 ($y = 0$).

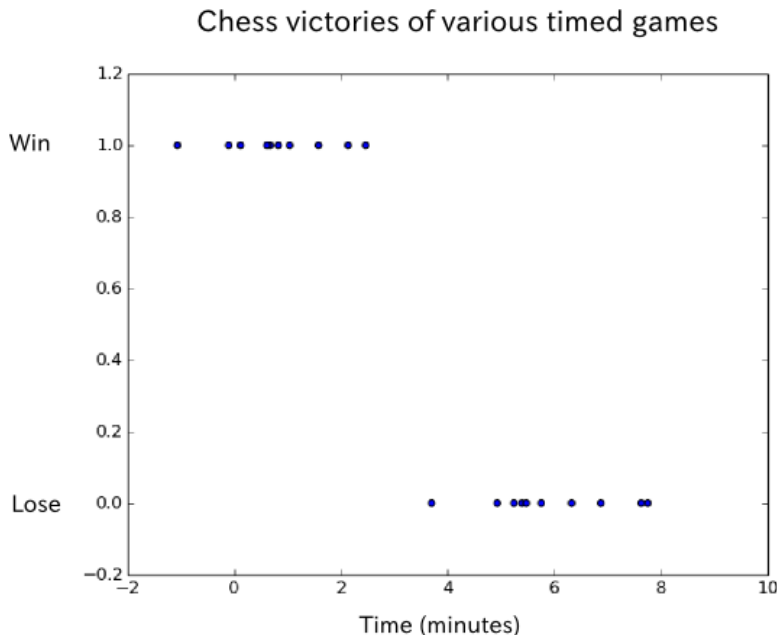


Figure 4.6 A visualization of a binary classification training dataset. The values are divided into two classes: all points where $y = 1$, and all points where $y = 0$.

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.

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.

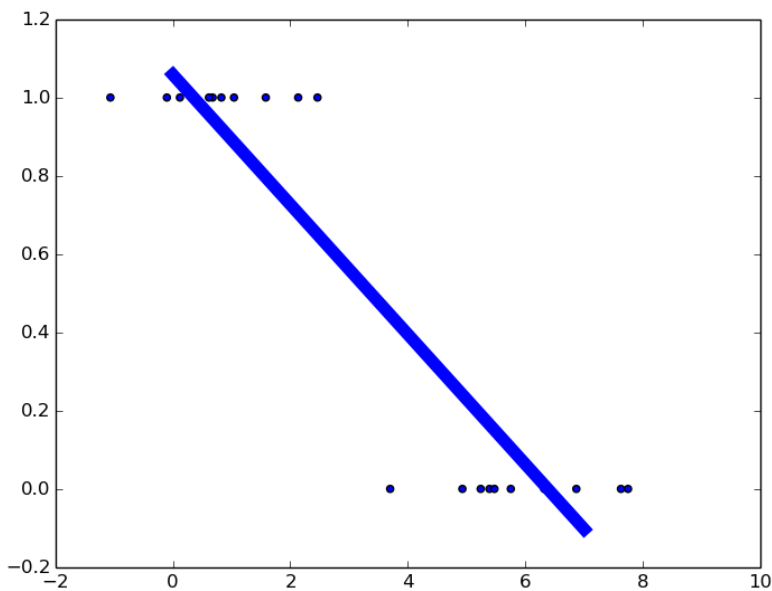


Figure 4.7 The diagonal line is the best fit line on a classification dataset. Clearly the line doesn't fit the data well, but it provides an out-of-date approach to classify new data.

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). Are you've seen, this procedure uses linear regression to perform classification.

EXERCISE 4.X What are the disadvantages of using linear regression as a tool for classification? (See listing 4.4 for a hint)

Let's write our first classifier! Open a new Python source file, and call it `linear.py`. Follow the code in listing 4.1 to write the code. 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 ##A
import numpy as np ##A
import matplotlib.pyplot as plt ##A

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

plt.scatter(xs, labels) ##D

learning_rate = 0.001 ##E
training_epochs = 1000 ##E

X = tf.placeholder("float") ##F
Y = tf.placeholder("float") ##F

def model(X, w): ##G
    return tf.add(tf.multiply(w[1], tf.pow(X, 1)), ##G
                  tf.multiply(w[0], tf.pow(X, 0))) ##G

w = tf.Variable([0., 0.], name="parameters") ##H
y_model = model(X, w) ##I
cost = tf.reduce_sum(tf.square(Y-y_model)) ##J

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

```
#A Import TensorFlow for the core learning algorithm, NumPy for manipulating data, and Matplotlib for visualizing
#B Initialize fake data, 10 instances of each label
#C Initialize the corresponding labels
#D Plot the data
#E Declare the hyper-parameters
#F Set up the placeholder nodes for the input/output pairs
#G Define a linear  $y = w_1 * x + w_0$  model.
#H Set up the parameter variables
#I Define a helper variable because we'll refer to this multiple times
#J Define the cost function
#K 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 generates a plot similar to figure 4.7.

Listing 4.2 Executing the graph

```
sess = tf.Session() ##L
```

```

init = tf.global_variables_initializer() //#L
sess.run(init) //#L

for epoch in range(training_epochs): //#M
    sess.run(train_op, feed_dict={X: xs, Y: labels}) //#M
    current_cost = sess.run(cost, feed_dict={X: xs, Y: labels}) //#N
    if epoch % 100 == 0:
        print(epoch, current_cost) //#O

w_val = sess.run(w) //#P
print('learned parameters', w_val) //#P

sess.close() //#Q

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

```

```

#L Open a new session and initialize the variables
#M Run the learning operation multiple times
#N Record the cost computed with the current parameters
#O Print out log info while the code runs
#P Print the learned parameters
#Q Close the session when no longer in use
#R 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))) //#A
accuracy = tf.reduce_mean(tf.to_float(correct_prediction)) //#B

print('accuracy', sess.run(accuracy, feed_dict={X: xs, Y: labels})) //#C

#A When the model's response is greater than 0.5, it should be a positive label, and vice versa
#B Compute the percent of success
#C 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_label10 = np.append(np.random.normal(5, 1, 9), 20)
```

When you re-run the code with the changes made in listing 4.4, you will see a result similar to figure 4.8.

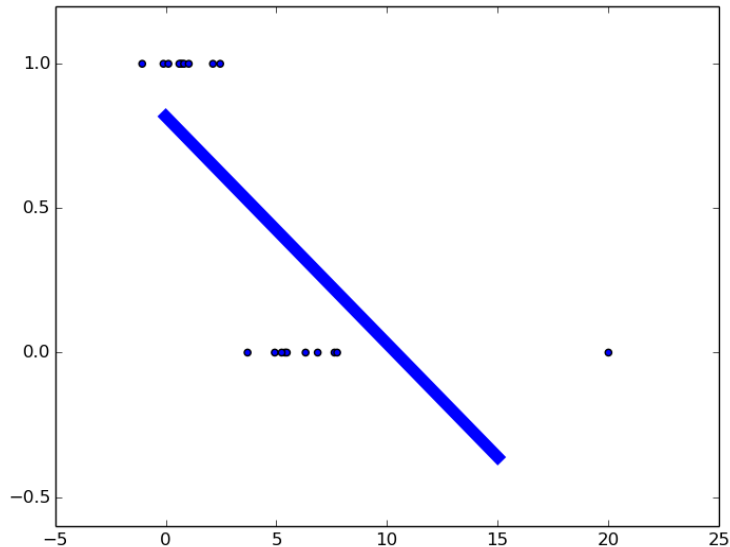


Figure 4.8 A new training element of value 20 greatly influences the best-fit line. The line is too sensitive to outlying data, and therefore linear regression is a sloppy classifier.

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.

$$y(x) = wx$$

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.

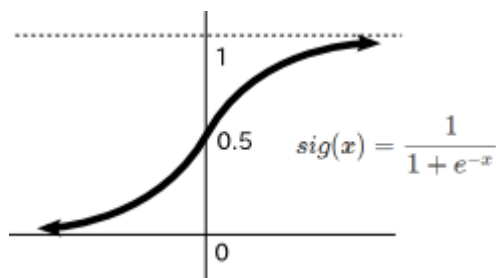


Figure 4.9 A visualization of the sigmoid function.

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 imply 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 from the one we used in linear regression. 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.

$$Cost(y, h) = \begin{cases} -\log(h), & \text{if } y = 1 \\ -\log(1 - h), & \text{if } y = 0 \end{cases}$$

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

$$\text{Cost}(y, h) = -y \log(h) - (1 - y) \log(1 - h)$$

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 intended it to do? To answer that question most intuitively, take a look at figure 4.10. We use $-\log(x)$ to compute the cost when we want our desired value to be 1 (notice: $-\log(1) = 0$). The algorithm strays away from setting the value to 0, since the cost approaches infinity. Adding these functions together gives a curve that approaches infinity at both zero and 1, with the negative parts cancelling out.

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.

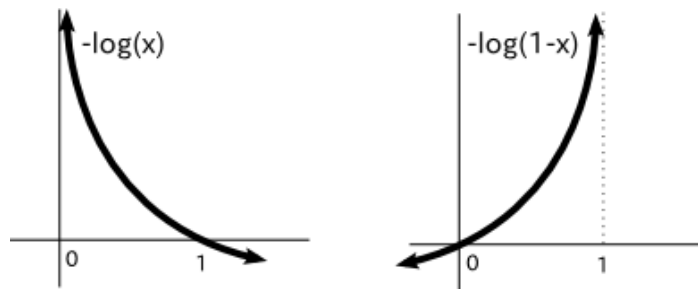


Figure 4.10 Here's a visualization of how the two different cost functions penalize values at 0 and 1. Notice how the left function heavily penalizes 0, but has no cost at 1. The right cost function displays the opposite phenomena.

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.

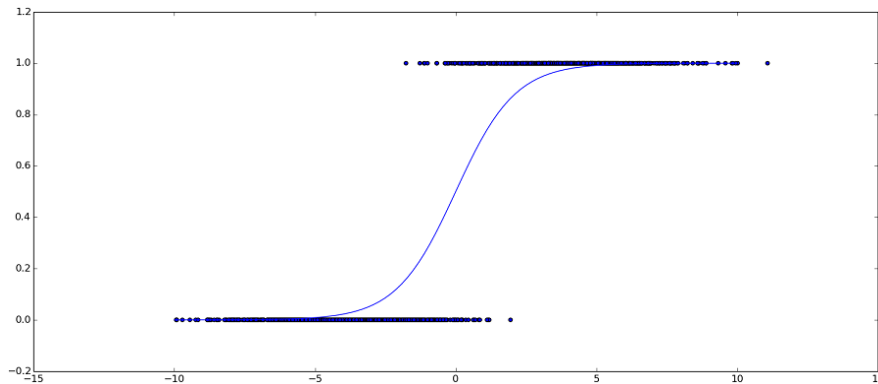


Figure 4.11 Here is a best fit sigmoid curve for a binary classification dataset. Notice how the curve resides within $y = 0$ and $y = 1$. That way, this curve is not that sensitive to outliers.

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  ##A
import tensorflow as tf  ##A
import matplotlib.pyplot as plt  ##A

learning_rate = 0.01  ##B
training_epochs = 1000  ##B

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

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

plt.scatter(xs, ys)  ##E

X = tf.placeholder(tf.float32, shape=(None,), name="x")  ##F
Y = tf.placeholder(tf.float32, shape=(None,), name="y")  ##F
w = tf.Variable([0., 0.], name="parameter", trainable=True)  ##G
y_model = tf.sigmoid(w[1] * X + w[0])  ##H
cost = tf.reduce_mean(-Y * tf.log(y_model) - (1 - Y) * tf.log(1 - y_model))  ##I

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



```

with tf.Session() as sess: //#K
    sess.run(tf.global_variables_initializer()) //#K
    prev_err = 0 //#L
    for epoch in range(training_epochs): //#M
        err, _ = sess.run([cost, train_op], {X: xs, Y: ys}) //#N
        print(epoch, err)
        if abs(prev_err - err) < 0.0001: //#O
            break
        prev_err = err //#P
    w_val = sess.run(w, {X: xs, Y: ys}) //#Q

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

```

```

#A Import relevant libraries
#B Set the hyper-parameters
#C Define a helper function to calculate the sigmoid function
#D Initialize fake data
#E Visualize the data
#F Define the input/output placeholders
#G Define the parameter node
#H Define the model using TensorFlow's sigmoid function
#I Define the cross-entropy loss function
#J Define the minimizer to use
#K Open a session and define all variables
#L Define a variable to track of the previous error
#M Iterate until convergence or until maximum number of epochs reached
#N Compute the cost as well as update the learning parameters
#O Check for convergence - if we're changing by < .01% per iteration, we're done
#P Update the previous error value
#Q Obtain the learned parameter value
#R Plot the learned sigmoid function

```

Cross-entropy loss in TensorFlow

As shown in listing 4.5, the cross-entropy loss is averaged over each input/output pair using the `tf.reduce_mean` op. Another handy and more general function is provided by the TensorFlow library, called `tf.nn.softmax_cross_entropy_with_logits`. You can find more about it in the official documentation: https://www.tensorflow.org/api_docs/python/tf/nn/softmax_cross_entropy_with_logits.

And there you have it! If you were playing chess against 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).

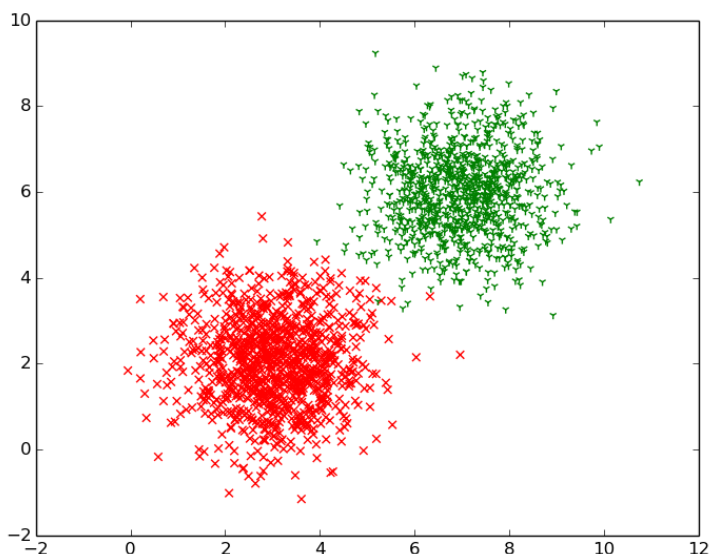


Figure 4.12 The x- and y-axis represent the two independent variables. The dependent variable holds two possible labels, represented by the shape and color of the plotted points.

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

Listing 4.6 Setting up data for two-dimensional logistic regression

```
import numpy as np //#A
import tensorflow as tf //#A
import matplotlib.pyplot as plt //#A

learning_rate = 0.1 //#B
training_epochs = 2000 //#B
```

```
def sigmoid(x):    ##C
    return 1. / (1. + np.exp(-x))    ##C

x1_label1 = np.random.normal(3, 1, 1000)    ##D
x2_label1 = np.random.normal(2, 1, 1000)    ##D
x1_label2 = np.random.normal(7, 1, 1000)    ##D
x2_label2 = np.random.normal(6, 1, 1000)    ##D
x1s = np.append(x1_label1, x1_label2)    ##D
x2s = np.append(x2_label1, x2_label2)    ##D
ys = np.asarray([0.] * len(x1_label1) + [1.] * len(x1_label2))    ##D
```

We have two independent variables (x_1 and x_2). A simple way to model the mapping between the input x 's and output $M(x)$ is the following equation, where w is the parameters to be found using TensorFlow.

$$M(x; w) = \text{sig}(w_2x_2 + w_1x_1 + w_0)$$

In listing 4.7, you will be implementing the equation and its corresponding cost function to learn the parameters.

Listing 4.7 Using TensorFlow for multi-dimensional logistic regression

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

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

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

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

plt.scatter(x1_boundary, x2_boundary, c='b', marker='o', s=20)    ##N
plt.scatter(x1_label1, x2_label1, c='r', marker='x', s=20)    ##N
```

```
plt.scatter(x1_label2, x2_label2, c='g', marker='1', s=20) // #N
plt.show() // #N
```

```
#A Import relevant libraries
#B Set the hyper-parameters
#C Define a helper sigmoid function
#D Initialize some fake data
#E Define the input/output placeholder nodes
#F Define the parameter node
#G Define the sigmoid model using both input variables
#H Define the learning step
#I Create a new session, initialize variables, and learn parameters until convergence
#J Obtain the learn parameter value before closing the session
#K Define arrays to hold boundary points
#L Loop through a window of points
#M If the model response is close the 0.5, then update the boundary points
#N 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.

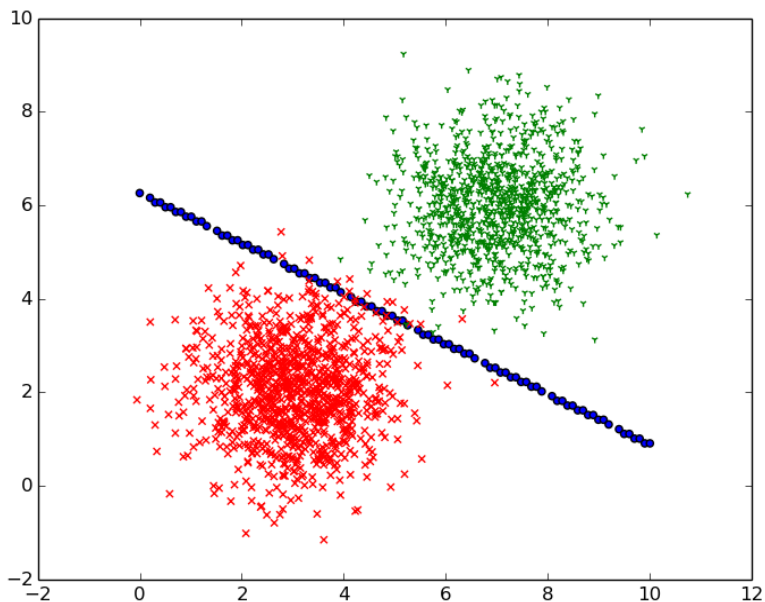


Figure 4.13 The diagonal dotted line represents when the probability between the two decisions is split equal. The confidence of making a decision increase as data lies further away from the line.

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 logistic regression approaches require a decent amount of ad-hoc engineering, so we'll focus our efforts on softmax regression.

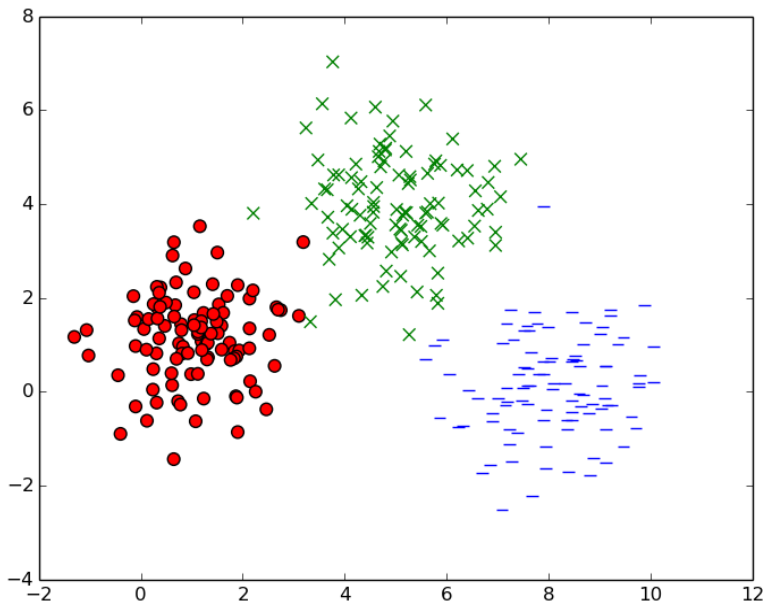


Figure 4.14 The independent variable is two-dimensional, indicated by the x and y-axis. The dependent variable can be one of three labels, shown by the color and shape of the datapoints.

4.5.1 One versus all

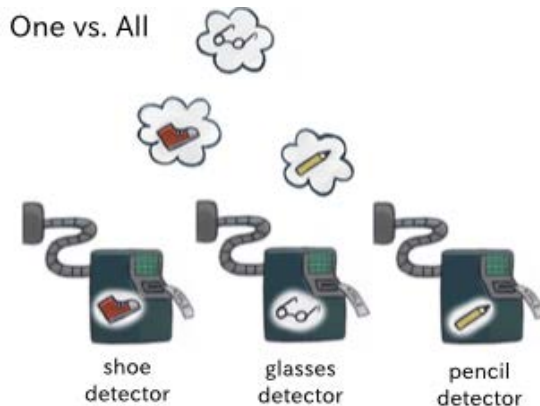


Figure 4.15 One versus All is a multi-class classifier approach that requires a detector for each class.

First, we train a classifier for each of the labels as visualized in figure 4.15. If there are three labels, we have three classifiers available to use: f_1 , f_2 , and f_3 . 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

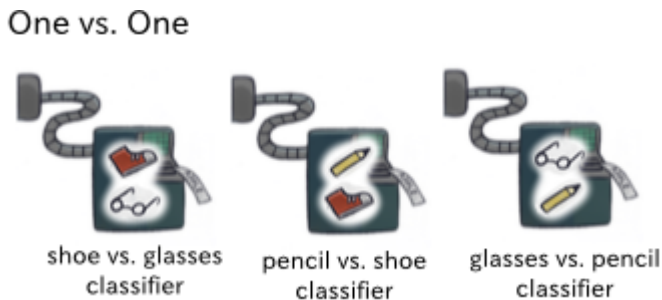


Figure 4.16 In One vs. One multi-class classification, there's a detector for each pair of classes.

Then 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 4.8 closely. First, we will visualize some fake data to reproduce figure 4.14 (also reproduced in figure 4.16).

Listing 4.8 Visualizing multiclass data

```
import numpy as np    ##A
import matplotlib.pyplot as plt    ##A

x1_label0 = np.random.normal(1, 1, (100, 1))    ##B
x2_label0 = np.random.normal(1, 1, (100, 1))    ##B
x1_label1 = np.random.normal(5, 1, (100, 1))    ##C
x2_label1 = np.random.normal(4, 1, (100, 1))    ##C
x1_label2 = np.random.normal(8, 1, (100, 1))    ##D
x2_label2 = np.random.normal(0, 1, (100, 1))    ##D

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

```
#A Import NumPy and Matplotlib
#B Generate points near (1, 1)
#C Generate points near (5, 4)
#D Generate points near (8, 0)
#E Visualize the three labels on a scatter plot
```

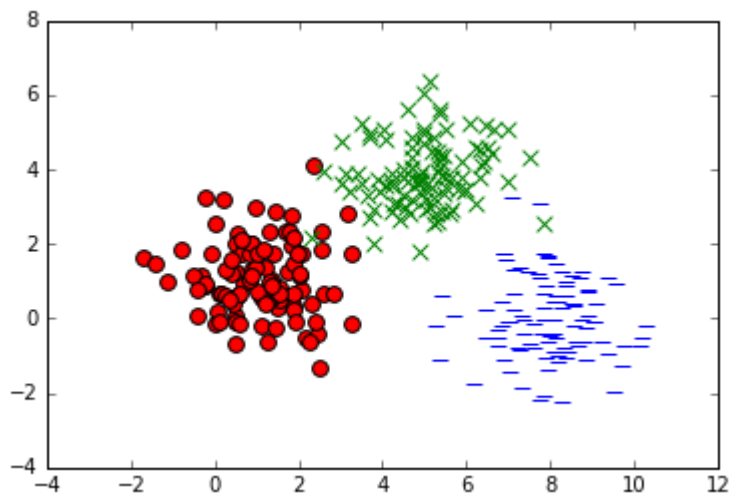


Figure 4.16 2D Training data for multi-output classification.

Next, in listing 4.9 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 4.4 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 use just 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 4.9 Setting up training and test data for multiclass classification

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

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

arr = np.arange(xs.shape[0]) //#C
np.random.shuffle(arr) //#C
xs = xs[arr, :] //#C
```



```

labels = labels[arr, :] //#C

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

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

train_size, num_features = xs.shape //#E

```

```

#A Combine all input data into one big matrix
#B Create the corresponding one-hot labels
#C Shuffle the dataset
#D Construct the test dataset and labels
#E The shape of the dataset tells you the number examples and features per example

```

Finally, in listing 4.10 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). Continuous and smooth functions facilitate learning the correct weights of a neural network by back-propagation.

EXERCISE 4.X Which of the following functions is continuous? $f(x) = x^2$. $f(x) = \min(x, 0)$. $f(x) = \tan(x)$.

Listing 4.10 Using softmax regression

```

import tensorflow as tf

learning_rate = 0.01 //#A
training_epochs = 1000 //#A
num_labels = 3 //#A
batch_size = 100 //#A

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

W = tf.Variable(tf.zeros([num_features, num_labels])) //#C
b = tf.Variable(tf.zeros([num_labels])) //#C
y_model = tf.nn.softmax(tf.matmul(X, W) + b) //#D

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

correct_prediction = tf.equal(tf.argmax(y_model, 1), tf.argmax(Y, 1)) //#F

```

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float")) // #F

#A Define hyper-parameters
#B Define the input/output placeholder nodes
#C Define the model parameters
#D Design the softmax model
#E Set up the learning algorithm
#F 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 4.11 for running the optimizer in batches.

Listing 4.11 Executing the graph

```
with tf.Session() as sess: // #H
    tf.global_variables_initializer().run() // #H

    for step in range(training_epochs * train_size // batch_size): // #I
        offset = (step * batch_size) % train_size // #J
        batch_xs = xs[offset:(offset + batch_size), :] // #J
        batch_labels = labels[offset:(offset + batch_size)] // #J
        err, _ = sess.run([cost, train_op], feed_dict={X: batch_xs, Y: batch_labels}) // #K
        print(step, err) // #L

    W_val = sess.run(W) // #M
    print('w', W_val) // #M
    b_val = sess.run(b) // #M
    print('b', b_val) // #M
    print("accuracy", accuracy.eval(feed_dict={X: test_xs, Y: test_labels})) // #N

#H Open a new session and initialize all variables
#I Loop only enough times to complete a single pass through the dataset
#J Retrieve a subset of the dataset corresponding to the current batch
#K Run the optimizer on this batch
#L Print on-going results
#M Print final learned parameters
#N Print success rate
```

The final output of running the softmax regression algorithm on our dataset results 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
```

We've learned the weights and biases of the model. We can reuse these learned parameters to infer on test data. A simple way to do so is by saving and loading the variables in TensorFlow's (see https://www.tensorflow.org/programmers_guide/variables). You can run the

model (which is called `y_model` in our code) to obtain the model responses on your test input data.

4.6 Application of classification

Emotion is a difficult concept to operationalize. Happiness, sadness, anger, excitement, and fear are some examples of emotions that are very subjective. What comes across as exciting to someone might appear sarcastic to another. Text that appears to convey anger to some might instead convey fear to others. If humans have so much trouble, what luck can computers have?

At the very least, machine learning researchers have figured out ways to classify positive and negative sentiment within text. For example, let's say you're building an Amazon-like website where each item has user reviews. You want your intelligent search engine to prefer items with positive reviews. Perhaps the best metric you have available is the average star rating or number of thumbs-ups. But what if you have a lot of heavy-text reviews without explicit ratings?

Sentiment analysis can be considered a binary classification problem. The input is some natural language text and the output is a binary decision inferring positive or negative sentiment. The following are some datasets you can find online to solve this exact problem.

- Large Movie Review Dataset:
 - <http://ai.stanford.edu/~amaas/data/sentiment/>
- Sentiment Labelled Sentences Data Set
 - <https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences>
- Twitter Sentiment Analysis Dataset
 - <http://thinknook.com/twitter-sentiment-analysis-training-corpus-dataset-2012-09-22/>

The biggest hurdle is to figure out how to represent raw text as an input to a classification algorithm. Throughout this chapter, the input to classification has always been a feature-vector. One of the oldest methods of converting raw text into a feature vector is called *bag-of-words*. You can find a very nice tutorial and code implementation for it here: <https://www.kaggle.com/c/word2vec-nlp-tutorial/details/part-1-for-beginners-bag-of-words>.

4.7 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 fiddle with more hyperparameters.

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.

Clustering is the process of intelligently categorizing the items in your dataset. The overall idea 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 the similarity of two species in the hierarchy of biological classification (family, genus, and species).

You can imagine 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.

First, we'll cover how to load audio files into TensorFlow and represent them as feature vectors. Then, we'll implement various clustering techniques to solve real-world problems.

5.1 Traversing files in TensorFlow

Some common input types in machine learning algorithms are audio and image files. This shouldn't come as a surprise, because sound recordings and photographs are raw, redundant, and often noisy representations of semantic concepts. Machine learning is a tool to help handle these complications.

These data files have various implementations: for example, 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.

EXERCISE 5.1 What are the pros and cons of MP3 and WAV? How about PNG vs. JPEG?

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.

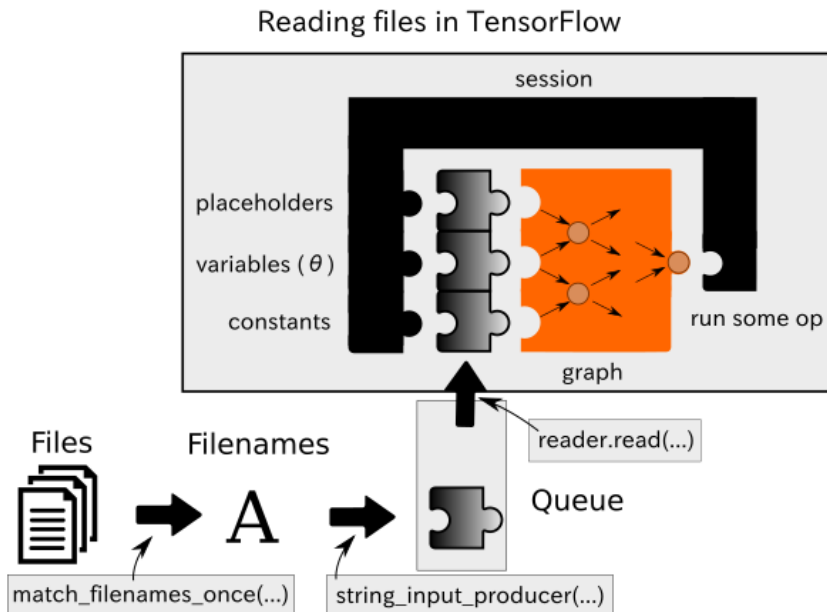


Figure 5.1 You can use a queue in TensorFlow to read files. The queue is built into the TensorFlow framework, and you can use the `reader.read(...)` function to access (and dequeue) it.

See listing 5.1 for 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') //#A
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames) //#B
reader = tf.WholeFileReader() //#C
filename, file_contents = reader.read(filename_queue) //#D

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_files = sess.run(count_num_files) //#E

    coord = tf.train.Coordinator() //#F
    threads = tf.train.start_queue_runners(coord=coord) //#F
```

```

for i in range(num_files): // #G
    audio_file = sess.run(filename) // #G
    print(audio_file) // #G

```

#A Store filenames that match a pattern
#B Set up a pipeline for retrieving filenames randomly
#C Natively read a file in TensorFlow
#D Run the reader to extract file data
#E Count the number of files
#F Initialize threads for the filename queue
#G Loop through the data one by one

RUNNING INTO PROBLEMS? If you couldn't get listing 5.1 to work, you may want to try the advice posted on this book's official forum: <https://forums.manning.com/posts/list/40966.page;jsessionid=182FC12469C6936E6BA100347D652592>

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. If you've ever seen a vinyl record, you've probably noticed the representation of audio as grooves indented in the disk. Our ears interpret audio from a series of vibrations through air. By recording the vibration properties, our algorithm can store sound in a data format.

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 implemented using an algorithm known as the 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
```

Python 2 Required

The BregmanToolkit is officially supported on Python 2. If you're using Jupyter notebook, you can have access to both version of Python by following directions outlined on the official Jupyter docs:

https://ipython.readthedocs.io/en/latest/install/kernel_install.html#kernels-for-python-2-and-3

In particular, you can include Python 2 with the following commands:


```
$ python2 -m pip install ipykernel
$ python2 -m -ipykernel install --user
```

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 $10 \times t$ 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):  //#A
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)  //#B
    return F.X  //#C
```

#A Pass in the filename

#B Use these parameters to describe 12 pitches every 0.1 seconds

#C 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.

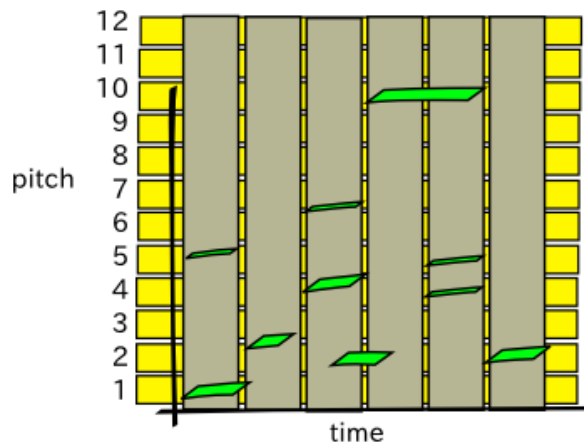


Figure 5.2 Visualization of the chromogram matrix where the x-axis represents time and the y-axis represents pitch class. The green markings indicate a presence of that pitch at that time.

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.

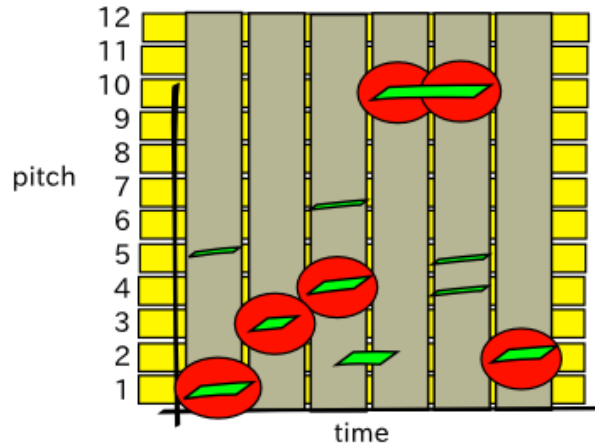


Figure 5.3 The most influential pitch at every time interval is highlighted. You can think of it as the loudest pitch at each time interval.

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.

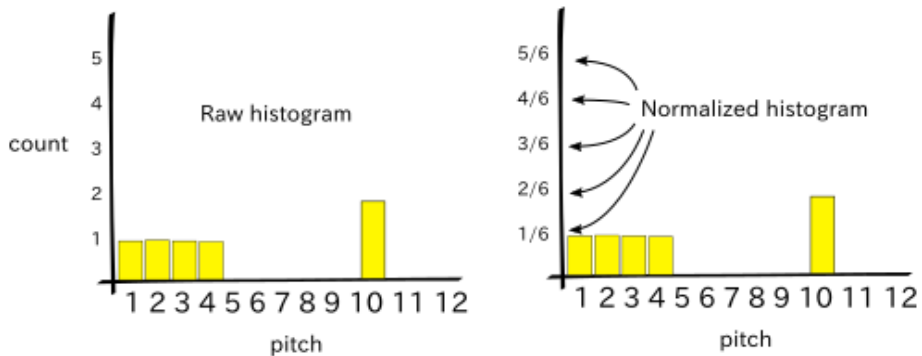


Figure 5.4 We count the frequency of loudest pitches heard at each interval to generate this histogram, which acts as our feature vector.

EXERCISE 5.2 What are some other ways to represent an audio clip as a feature vector?

Let's take a look at listing 5.3 to generate this histogram, which is our feature vector.

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) //#A
max_freqs = tf.argmax(chromo, 0) //#A

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): #B
    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): #C
    num_files = sess.run(count_num_files)
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
    xs = []
    for _ 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
```

#A Create an op to identify the pitch with the biggest contribution

#B Convert a chromogram into a feature vector

#C Construct a matrix where each row is a data item

BY THE WAY All code listing are available on GitHub https://github.com/BinRoot/TensorFlow-Book/tree/master/ch05_clustering

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. In simplified terms, a straightforward way of determining the best value of k is to simply iterate over a range of k -means simulations and apply a cost function to determine which value of k caused the best differentiation between clusters at the lowest value of k .

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.

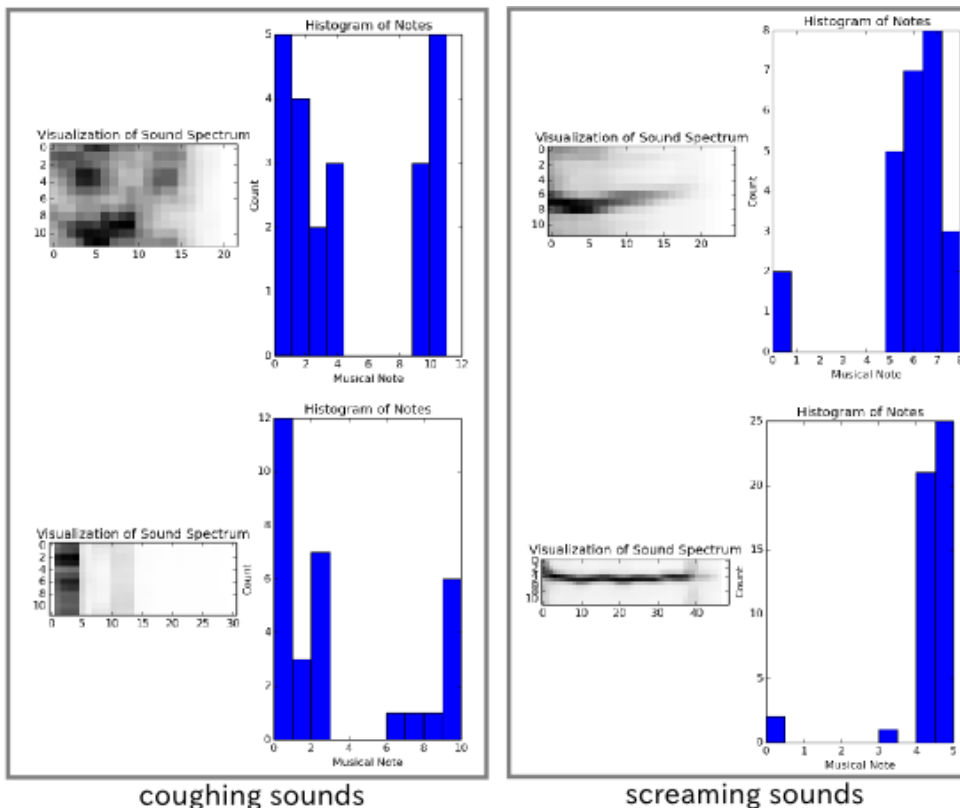


Figure 5.5 Here are four examples of audio files. As you can see, the two on the right appear to have similar histograms. The two on the left also have similar histograms. Our clustering algorithms will be able to group these sounds together.

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. See figure 5.6 for a visualization of the algorithm.

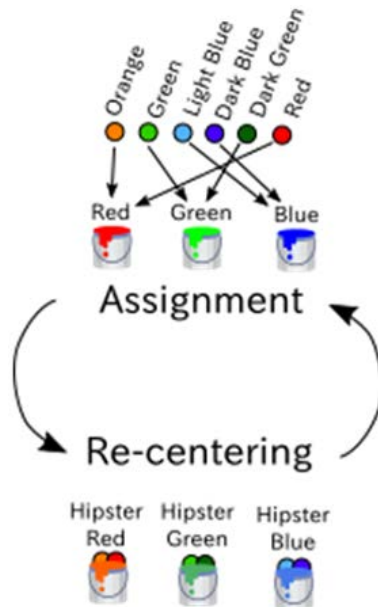


Figure 5.6 This figure shows one iteration of the k-means algorithm. Let's say we're clustering colors into 3 buckets (which is an informal way to say "category"). We can start with initial guess of red, green, and blue and begin the assignment step as shown above. Then we update the bucket colors by averaging the colors that belong to each bucket. We keep repeating until the buckets no longer substantially change color, arriving at the color representing the centroid of each cluster

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 #A
max_iterations = 100 #B

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

def assign_cluster(X, centroids): #D
    expanded_vectors = tf.expand_dims(X, 0)
    expanded_centroids = tf.expand_dims(centroids, 1)
    distances = tf.reduce_sum(tf.square(tf.subtract(expanded_vectors, expanded_centroids)),
        2)
    mins = tf.argmin(distances, 0)
    return mins
```

```

def recompute_centroids(X, Y): #E
    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.global_variables_initializer())
    X = get_dataset(sess)
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations: #F
        i += 1
        Y = assign_cluster(X, centroids)
        centroids = sess.run(recompute_centroids(X, Y))
    print(centroids)

```

#A Decide the number of clusters

#B Declare the maximum number of iterations to run k-means

#C Choose the initial guesses of cluster centroids

#D Assign each data item to its nearest cluster

#E Update the cluster centroids to their midpoint

#F 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 5.7.

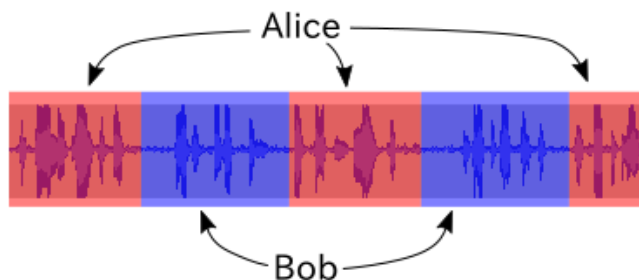


Figure 5.7 Audio segmentation is the process of automatically labelling segments.

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 would 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 = 2 #A
segment_size = 50 #B
max_iterations = 100 #C

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): #D
    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
```

#A Decide the number of clusters

#B The smaller the segment size, the better the results (but slower performance)

#C Decide when to stop the iterations

#D 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, 'TalkingMachinesPodcast.wav')
    print(np.shape(X))
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations: // #A
        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)): // #B
        seconds = (i * segment_size) / float(10)
        min, sec = divmod(seconds, 60)
        time_str = '{}m {}'.format(min, sec)
        print(time_str, segments[i])
```

#A Run the k-means algorithm

#B Print the labels for each time interval

The output of running listing 5.6 is a list of timestamps and cluster ids that correspond to who is talking during the podcast:

```
('0.0m 0.0s', 0)
('0.0m 2.5s', 1)
('0.0m 5.0s', 0)
('0.0m 7.5s', 1)
('0.0m 10.0s', 1)
('0.0m 12.5s', 1)
('0.0m 15.0s', 1)
('0.0m 17.5s', 0)
('0.0m 20.0s', 1)
('0.0m 22.5s', 1)
('0.0m 25.0s', 0)
('0.0m 27.5s', 0)
```

EXERCISE 5.3 How can you detect whether the clustering algorithm has converged (so that you can stop the algorithm early)?

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. An 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 5.8, in which the clusters change over time.



Figure 5.8 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 number of neighbors around a BMU shrinks over time at a rate determined usually by trial and error. See figure 5.9 for a visualization of the algorithm.

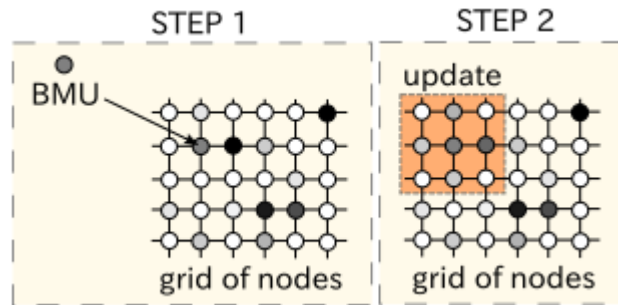


Figure 5.9 This figure visualizes one iteration of the SOM algorithm. The first step is to identify the best matching unit (BMU), and the second step is to update the neighboring nodes. We keep iterating these two steps with training data until some convergence criteria.

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])) //#A
        self.nodes = nodes

        x = tf.placeholder(tf.float32, [dim]) //#B
        iter = tf.placeholder(tf.float32) //#B

        self.x = x //#C
        self.iter = iter //#C

        bmu_loc = self.get_bmu_loc(x) //#D

        self.propagate_nodes = self.get_propagation(bmu_loc, x, iter) //#E
```

#A Each node is a vector of dimension `dim`. For a 2D grid, there are `width * height` nodes; `get_locs` is defined in Listing 5.10 below
 #B These two ops are inputs at each iteration
 #C We'll need to access them from another method
 #D Find the node that matches closest to the input; in Listing 5.9 below
 #E Update the values of the neighbors; in Listing 5.8 below

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) // #A
    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) // #B
    sqr_dists_from_bmu = tf.reduce_sum(
        tf.square(tf.subtract(expanded_bmu_loc, self.node_locs)), 1)
    neigh_factor = // #C
        tf.exp(-tf.div(sqr_dists_from_bmu, 2 * tf.square(sigma)))
    rate = tf.multiply(alpha, neigh_factor)
    rate_factor =
        tf.stack([tf.tile(tf.slice(rate, [i], [1]),
            [self.dim]) for i in range(num_nodes)])
    nodes_diff = tf.multiply(
        rate_factor,
        tf.subtract(tf.stack([x for i in range(num_nodes)]), self.nodes))
    update_nodes = tf.add(self.nodes, nodes_diff) // #D
    return tf.assign(self.nodes, update_nodes) // #E
```

#A The rate decreases as as iter increases. This value influences alpha and sigma parameters.
 #B Expand bmu_loc, so we can efficiently compare it pairwise with each element of node_locs
 #C Ensure that nodes closer to the bmu charge more dramatically
 #D Define the updates
 #E Return an op to perform the updates

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.subtract(expanded_x, self.nodes))
    dists = tf.reduce_sum(sqr_diff, 1)
    bmu_idx = tf.argmin(dists, 0)
    bmu_loc = tf.stack([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 `global_variables_initializer` op. Next, we loop `num_iters` some 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.global_variables_initializer())
        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 5.10 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) // #A
som.train(colors)
```

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

#A The grid size is 4x4, and the input dimension is 3

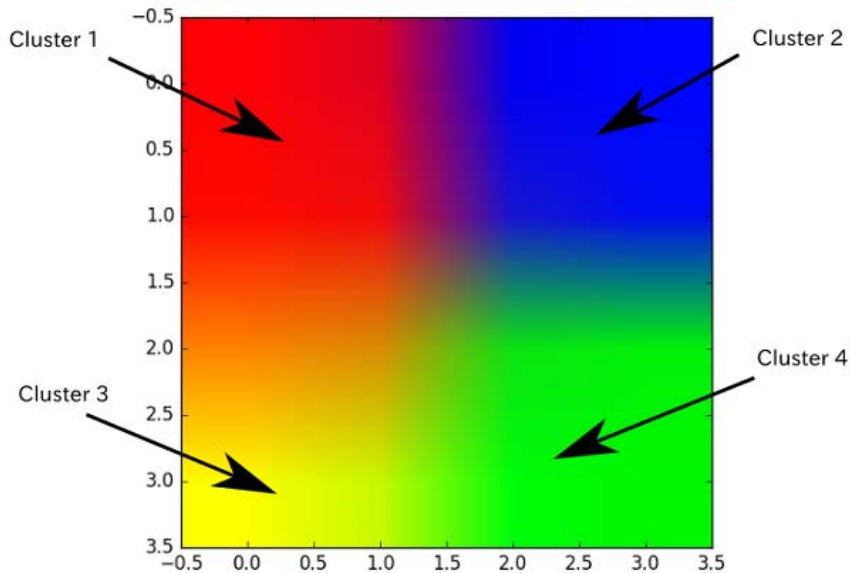


Figure 5.10 Here's a visualization of the SOM's output. It placed all 3-dimensional data points into a 2-dimensional grid. From it, you can pick the cluster centroids (automatically or manually) and achieve clustering in an intuitive lower-dimensional space.

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. For the adventurous, I suggest reading the famous paper by Juha Vesanto and Esa Alhoniemi on Clustering SOMs: <http://lib.tkk.fi/Diss/2002/isbn951226093X/article4.pdf>.

5.6 Application of clustering

We've already seen two practical applications of clustering, namely how it can help organize music and how it can segment an audio clip to label similar sounds.

Clustering is especially helpful when the training dataset does not contain corresponding labels. As you know, such a situation characterizes unsupervised learning. Sometimes, data is just too inconvenient to annotate.

For example, suppose you want to understand sensor data from the accelerometer of a phone or smartwatch. At each time-step, the accelerator provides a 3-dimensional vector, but you have no idea whether the human is walking, standing, sitting, dancing, jogging, or so on.

You can obtain such a dataset at

<http://archive.ics.uci.edu/ml/datasets/User+Identification+From+Walking+Activity>.

To cluster the time-series data, we'll need to summarize the list of accelerator vectors into a concise feature vector. One way is to generate a histogram of differences between consecutive magnitudes of the acceleration. The derivative of acceleration is called *jerk*, and we can apply the same operation to obtain the histogram outlining difference in jerk magnitudes.

This process of generating a histogram out of data is exactly like the pre-processing steps on audio data explained in this chapter. Once you have transformed the histograms into feature-vectors, you can use the same code listings taught earlier (such as k-means in TensorFlow).

5.7 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 of 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 (SOM) 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

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 6.1 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 overly 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 6.2.

Before going into details about Markov chains and HMMs, let's consider some alternative models. Follow along to section 6.1 to witness how some models may not be interpretable.

6.1 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 may 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.

6.2 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 situations, or *states*, that we care to study. Figure 6.1 shows three weather states as nodes in a graph: cloudy, rainy, and sunny.

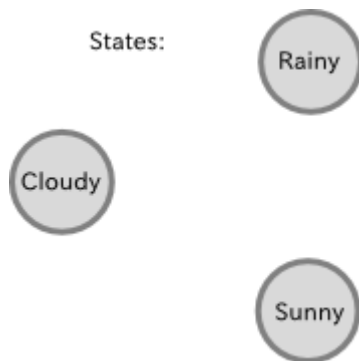


Figure 6.1 Weather conditions represented as nodes in a graph.

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 6.2 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 6.2 demonstrates the transitions as directed edges drawn between nodes, with the arrow pointing toward the next future state. Each edge has a weight representing the probability (for example, there's a 30% chance that if today is rainy, tomorrow will be cloudy). 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 $N \times N$ in size.

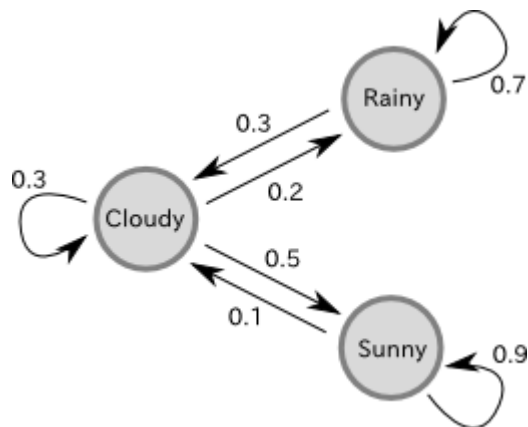


Figure 6.2 Transition probabilities between weather conditions are represented as directed edges.

We call this system a *Markov model*. Over time, a state changes using the transition probabilities previously defined (as shown in Figure 6.2). Figure 6.3 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.

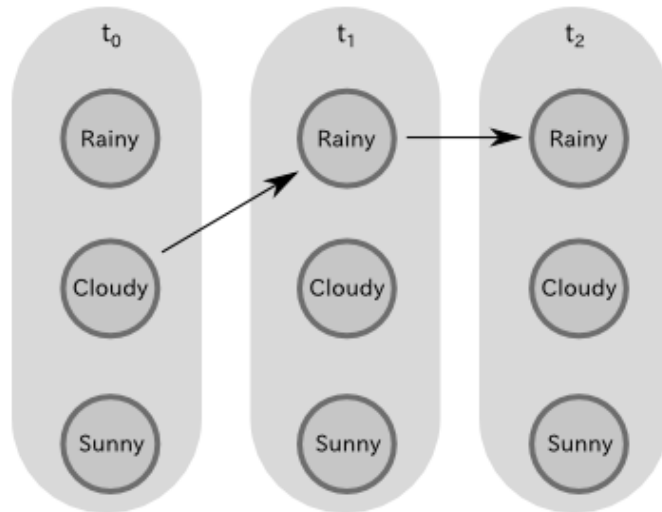








Figure 6.3 A trellis representation of the Markov system changing states over time.

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 6.2 and 6.3 nicely visualize state transitions, there's a more efficient way they to implement them in code, as shown in figure 6.4.

				
		Cloudy	Rainy	Sunny
	Cloudy	0.3	0.2	0.5
	Rainy	0.3	0.7	0.0
	Sunny	0.1	0.0	0.9

3x3 Transition Matrix

Figure 6.4 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 3×1 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 greatly simplify how we view the world. However, it's often difficult to measure the state of the world directly. Often, we have to use evidence from multiple observations to figure out the hidden meaning. And that's what the next section aims to solve!

6.3 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 temperature 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 6.5.

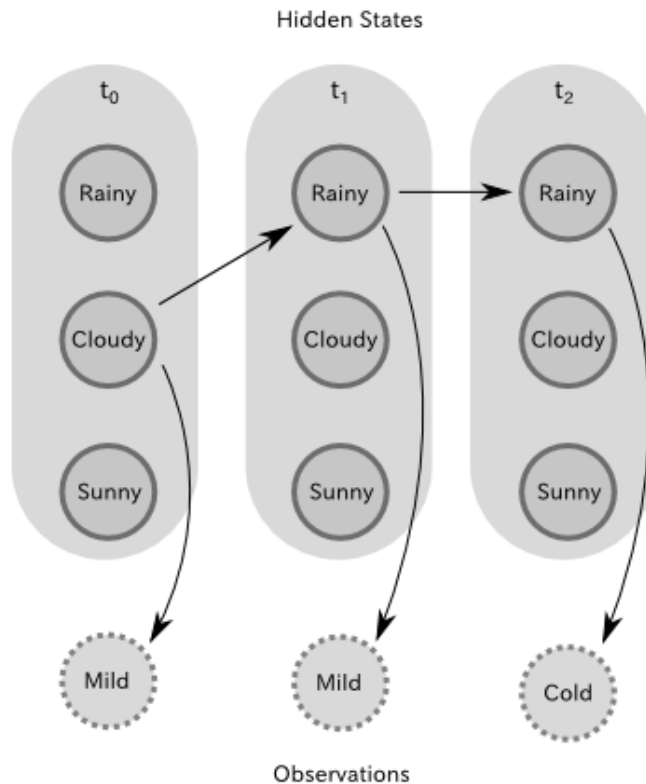


Figure 6.5 A Hidden Markov Model trellis showing how weather conditions might produce temperature readings.

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 modeling 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 modeling 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*.

6.4 Forward algorithm

The forward algorithm computes the probability of an observation. There are many permutations that may cause a particular observation, so enumerating all possibilities the naive 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 6.1, 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 6.1 Define the HMM class

```
import numpy as np ##A
import tensorflow as tf ##A

class HMM(object):
    def __init__(self, initial_prob, trans_prob, obs_prob):
        self.N = np.size(initial_prob) ##B
        self.initial_prob = initial_prob ##B
        self.trans_prob = trans_prob ##B
        self.emission = tf.constant(obs_prob) ##B

        assert self.initial_prob.shape == (self.N, 1) ##C
        assert self.trans_prob.shape == (self.N, self.N) ##C
        assert obs_prob.shape[0] == self.N ##C

        self.obs_idx = tf.placeholder(tf.int32) ##D
        self.fwd = tf.placeholder(tf.float64) ##D
```

#A Import the required libraries
 #B store the parameters as method variables
 #C double-check the shapes of all the matrices makes sense
 #D define the placeholders used for the forward algorithm

Next, we'll define a quick helper function in listing 6.2 to access a row from the emission matrix. The code in this listing is just a helper function to efficiently obtain data from an arbitrary matrix. The slice function extracts a fraction of the original tensor, as shown in listing 6.2. It requires as input the relevant tensor, the starting location specified by a tensor, and the size of the slice specified by a tensor.

Listing 6.2 Create a helper function to access emission probability of an observation

```
def get_emission(self, obs_idx):
    slice_location = [0, obs_idx] ##A
    num_rows = tf.shape(self.emission)[0]
    slice_shape = [num_rows, 1] ##B
    return tf.slice(self.emission, slice_location, slice_shape) ##C
```

#A The location of where to slice the emission matrix
 #B The shape of the slice
 #C Perform the slicing operator

We'll need to define two TensorFlow ops. The first one (in listing 6.3) will be run only once to initialize the forward algorithm's cache.

Listing 6.3 Initializing the cache

```
def forward_init_op(self):
    obs_prob = self.get_emission(self.obs_idx)
    fwd = tf.multiply(self.initial_prob, obs_prob)
    return fwd
```

And the next op will update the cache at each observation, as shown in listing 6.4. Running this code is often called executing a forward step. Although it looks like this `forward_op` function takes no input, it actually depends on placeholder variables that need to be fed to the session. Specifically, `self.fwd` and `self.obs_idx` are the inputs to this function.

Listing 6.4 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 6.5. The forward algorithm runs the forward step for each observation. In the end, it finally outputs a probability of observations.

Listing 6.5 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*. For consistency, the example in listing 6.6 below is lifted directly from the Wikipedia article on HMMs: https://en.wikipedia.org/wiki/Hidden_Markov_model#A_concrete_example, as shown in figure 6.6.


```

states = ('Rainy', 'Sunny')

observations = ('walk', 'shop', 'clean')

start_probability = {'Rainy': 0.6, 'Sunny': 0.4}

transition_probability = {
    'Rainy': {'Rainy': 0.7, 'Sunny': 0.3},
    'Sunny': {'Rainy': 0.4, 'Sunny': 0.6},
}

emission_probability = {
    'Rainy': {'walk': 0.1, 'shop': 0.4, 'clean': 0.5},
    'Sunny': {'walk': 0.6, 'shop': 0.3, 'clean': 0.1},
}

```

Figure 6.6 Screenshot of HMM example scenario from Wikipedia

In general, the three concepts are defined as follows.

- Initial probability vector: starting probability of the states.
- Transition probability matrix: probabilities associated with landing on the next states given the current state.
- Emission probability matrix: likelihood of an observed state implying the state we're interested in has occurred

Given these matrices, we'll call the forward algorithm that we just defined.

Listing 6.6 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([[0.1, 0.4, 0.5],
                         [0.6, 0.3, 0.1]])

    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 6.6, the algorithm will output the following:

```
Probability of observing [0, 1, 1, 2, 1] is 0.0045403
```

6.5 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 6.7.

Listing 6.7 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 6.8, let's define a TensorFlow op to update the `viterbi` cache. This will be a method in the HMM class.

Listing 6.8 Define an op to update the forward cache

```
def decode_op(self):
    transitions = tf.matmul(self.viterbi, tf.transpose(self.get_emission(self.obs_idx)))
    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 6.9 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 6.10, define the Viterbi decoding function outside of the HMM.

Listing 6.10

```
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 6.11 in the main function to evaluate the Viterbi decoding of an observation.

Listing 6.11 Run the Viterbi decode

```
seq = viterbi_decode(sess, hmm, observations)
print('Most likely hidden states are {}'.format(seq))
```

6.6 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.

6.6.1 Modeling 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 gait. Consider a HMM where the sequence of hidden states for a gait 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/>).

6.6.2 Modeling 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).

6.6.3 Modeling 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 characters are written in sequences – words - 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.

6.7 Application of hidden Markov models

Hidden Markov models work best when you have an intuition of what the hidden states are and how they change over time. Luckily, in the field of natural language processing, tagging the parts-of-speech of a sentence can be more or less solved using HMMs:

- A sequence of words in a sentence correspond to the observations of the HMM. For example, the sentence "Open the pod bay doors, HAL," has 6 observed words.
- The hidden states are the parts-of-speech, such as "verb," "noun," "adjective," and so

on. The observed word “open” in the previous example should correspond to the hidden state of “verb.”

- The transition probabilities can be designed by the programmer or obtained through data. Essentially, it represents the rules of parts-of-speech. For example, the probability of two verbs occurring one after another should be very low. By setting up a transition probability, the algorithm avoids brute-forcing all possibilities.
- The emitting probabilities of each word can be obtained from data. A traditional part-of-speech tagging dataset is called Moby, and can be found at <http://icon.shef.ac.uk/Moby/mpos.html>.

6.8 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

7

A peek into autoencoders



This chapter covers

- Introduction to neural networks
- Designing autoencoders
- Representing images using an autoencoder

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 a 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. As we've learned, our model is a function that gives the output we're looking for, given the inputs we have. In ML terms, given training data, we want to build a neural network model that best approximates the implicit function that might have generated the data – one that might not give us the exact answer, but one that is good enough to be useful.

So far, we've generated models by explicitly designing a function, whether it be linear, polynomial, or something more complicated. Neural networks enable a little bit of leeway when it comes to picking out the right function, and consequently the right model. In theory, a neural network can model general-purpose types of transformation – one where we don't need to know much about the function being modeled at all!

After introducing neural networks in section 7.1, we'll learn how to use autoencoders, which encode data into a smaller, faster representation, in section 7.2.

7.1 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 7.1.

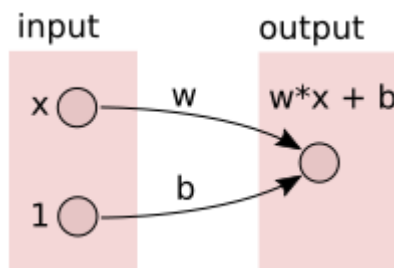


Figure 7.1 This is a graphical representation of a linear equation $f(x) = w * x + b$. The nodes are represented as circles, and edges are represented by arrows. The values on the edges are often called weights, and they act as a multiplication on the input. When two arrows lead to the same node, they act as a summation of the inputs.

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).

EXERCISE 7.1 Is $f(x) = |x|$ a linear function?

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 7.2.

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 7.2. Usually, the best one is chosen by using cross-validation to determine which one gives the best model given the dataset we're working with. Remember our confusion matrix in chapter 4? We test which model gives the fewest false-positives or least false-negatives, or whatever other criteria best suits our needs.

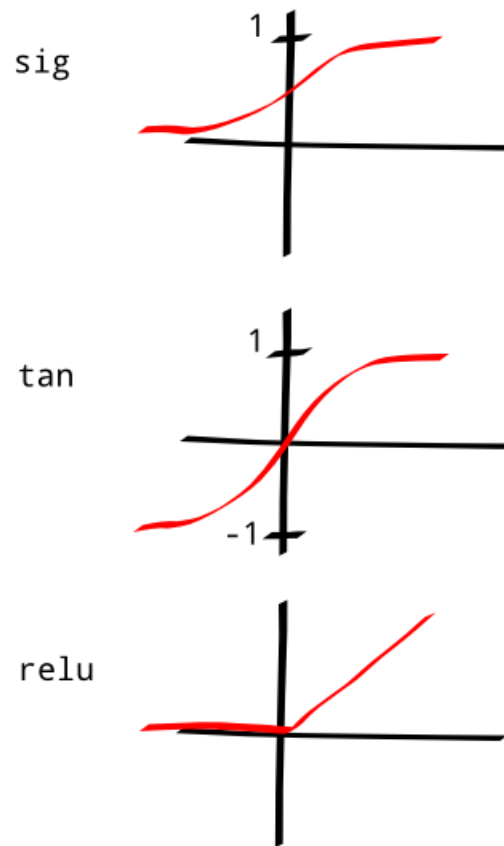


Figure 7.2 We use non-linear functions such as sig, tan, and relu to introduce non-linearity to our models.

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 7.3 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.

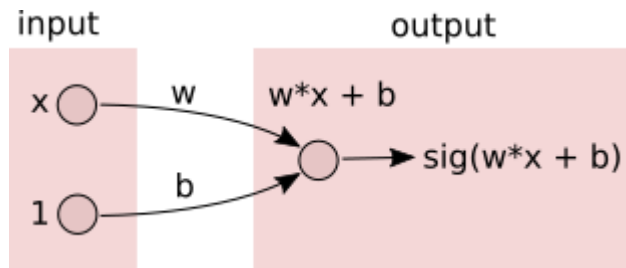


Figure 7.3 A non-linear function, such as sigmoid, is applied to the output of a node

If instead we have two inputs (x_1 and x_2), we can modify our neural network to look like the one in figure 7.4. Given training data and a cost function, the parameters to be learned are w_1 , w_2 , 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.

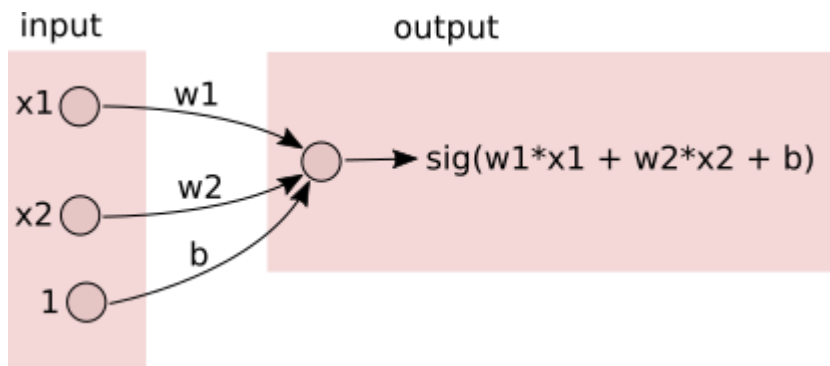


Figure 7.4 A 2-input network will have 3 parameters (w_1 , w_2 , and b). Remember, multiple lines leading to the same node indicate summation.

Naturally, we can generalize to an arbitrary number of inputs (x_1 , x_2 , ..., x_n). The corresponding neural network represents the function

$$f(x_1, \dots, x_n) = \text{sig}(w_n * x_n + \dots + w_1 * x_1 + b)$$

as seen in figure 7.5.

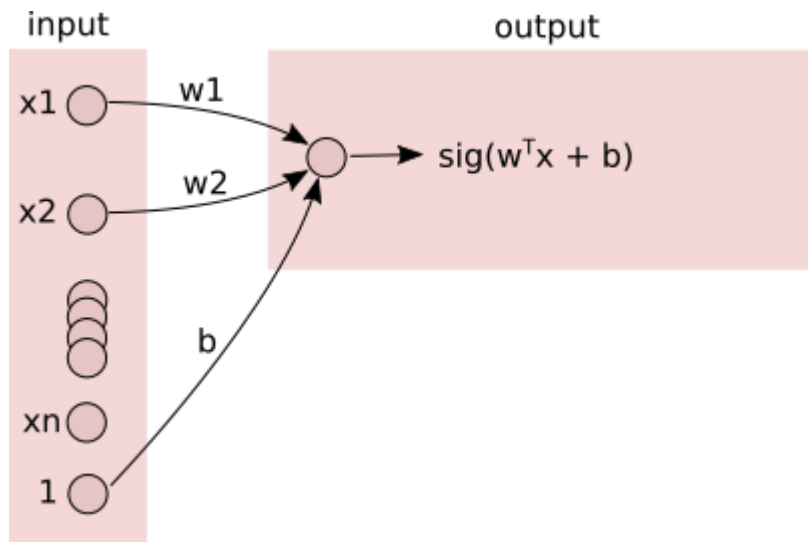


Figure 7.5 The input dimension can be arbitrarily long. For example, each pixel in a grayscale image can have a corresponding input x_i . This neural network uses all inputs to generate a single output number, which we might use for regression or classification. The notation w^T means we're transposing w , which is a $n \times 1$ vector, into a $1 \times n$ vector. That way, we can properly multiply it with x (which has the dimensions of $n \times 1$). Such a matrix multiplication is also called a dot product, and it yields a scalar (1-dimensional) value.

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*. They're hidden from the input and output interfaces of the neural network, so no one can directly influence their values. A hidden layer is any collection of hidden neurons that do not connect to each other, as seen in figure 7.6. Adding more hidden layers greatly improves the expressive power of the network.

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 non-linear neural network model with a hidden layer, on the other hand, is flexible enough to approximately represent any function! What a time to be alive!

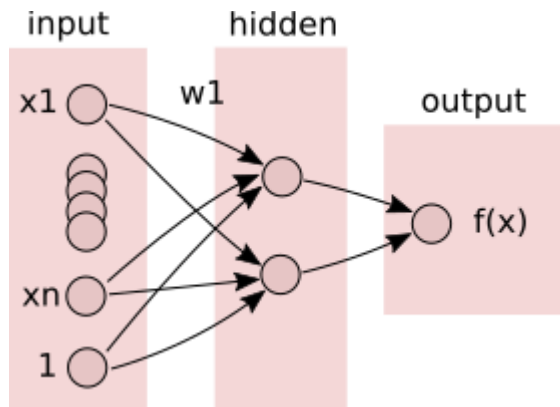


Figure 7.6 Nodes that do not interface to both the input and the output are called hidden units. A hidden layer is a collection of hidden units that are not connected to each other.

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.

7.2 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 7.7.

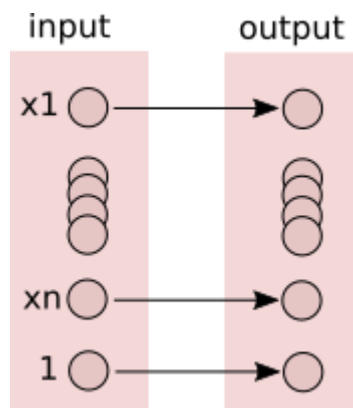


Figure 7.7 If we want to create a network where the input equals the output, we can just connect the corresponding nodes and set each parameter's weight to 1.

But an autoencoder is more interesting than that. It contains a small hidden layer! If that hidden layer has a smaller dimension than the input, the hidden layer is a compression of your data, called *encoding*.

Encoding data in the real-world

There's a couple different audio formats out there, but the most popular may be MP3 because of its relatively small file size. You may have already guessed that such efficient storage comes with a tradeoff. The algorithm to generate an MP3 file takes original uncompressed audio and shrinks it into a much smaller file that sounds approximately the same to your ears. However, it's *lossy*, meaning that you won't be able to completely recover the original uncompressed audio from the encoded version. Similarly in this chapter, we want to reduce the dimensionality of the data to make it more workable, but not necessarily a perfect reproduction.

The process of reconstructing the input from the hidden layer is called *decoding*. Figure 7.8 shows an exaggerated example of an autoencoder.

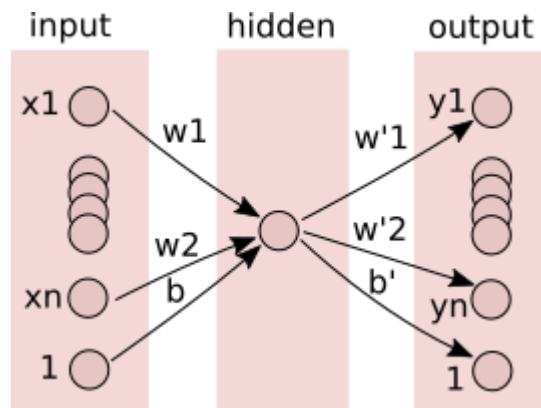


Figure 7.8 Here, we introduce a restriction to a network that tries to reconstruct its input. Data will pass through a narrow channel, as seen by the hidden layer. In this example, there is only 1 node in the hidden layer. So, this network is trying to encode (and decode) an n -dimensional input signal 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 thousands!

EXERCISE 7.2 Let x denote the input-vector (x_1, x_2, \dots, x_n), and let y denote the output-vector (y_1, y_2, \dots, y_n). Lastly, let w and w' denote the encoder and decoder weights. What is a possible cost function to train this neural network?

ANSWER See the loss function in Listing 7.3.

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 7.1 helps build deeper architectures, such as a *stacked autoencoder*, which has been known to perform 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 7.1 Python class schema

```
class Autoencoder:
    def __init__(self, input_dim, hidden_dim):
        #A

    def train(self, data):
        #B

    def test(self, data):
        #C
```

#A Initialize variables
#B Train on a dataset
#C 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 7.2 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 7.2 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 7.3.

Listing 7.3 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 //#A
        self.learning_rate = learning_rate //#B

        x = tf.placeholder(dtype=tf.float32, shape=[None, input_dim]) //#C

        with tf.name_scope('encode'): //#D
            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'): //#E
            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 //#F
        self.encoded = encoded //#F
        self.decoded = decoded //#F

        self.loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(self.x, self.decoded))))
        //#G
        self.train_op = tf.train.RMSPropOptimizer(self.learning_rate).minimize(self.loss)
        //#H
        self.saver = tf.train.Saver() //#I

```

#A Number of learning cycles

#B Hyper-parameter of optimizer

#C Define the input layer dataset

#D Define the weights and biases under a name scope so we can tell them apart from the decoder's weights and biases

#E The decoder's weights and biases are defined under this name scope

#F These will be method variables

#G Define the reconstruction cost

#H Choose the optimizer

#I Setup a saver to save model parameters as they're being learned

Now, in listing 7.3, we'll define a class method called `train` that will receive a dataset and learn parameters to minimize its loss.

Listing 7.3 Train the autoencoder

```

def train(self, data):
    num_samples = len(data)
    with tf.Session() as sess: //#A
        sess.run(tf.global_variables_initializer()) //#A
        for i in range(self.epoch): //#B
            for j in range(num_samples): //#C
                l, _ = sess.run([self.loss, self.train_op], feed_dict={self.x:
                [data[j]]}) //#C
            if i % 10 == 0: //#D
                print('epoch {0}: loss = {1}'.format(i, l)) //#D

```

```

        self.saver.save(sess, './model.ckpt') ##E
self.saver.save(sess, './model.ckpt') ##E

```

```

#A Start a TensorFlow session and initialize all variables
#B Iterate through the number of cycles defined in the constructor
#C One-by-one train the neural network on a data item
#D Print the reconstruction error once every 10 cycles
#E 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 7.4, the test method will let you evaluate the autoencoder on new data.

Listing 7.4 Test the model on some data

```

def test(self, data):
    with tf.Session() as sess:
        self.saver.restore(sess, './model.ckpt') ##A
        hidden, reconstructed = sess.run([self.encoded, self.decoded], feed_dict={self.x:
data}) ##B
        print('input', data)
        print('compressed', hidden)
        print('reconstructed', reconstructed)
    return reconstructed

```

```

#A Load the learned parameters
#B Reconstruct the input

```

Finally, let's create a new Python source file called `main.py` and use our Autoencoder class, as show in listing 7.5.

Listing 7.5 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]])

```

Running the `train` function will output some debug info about how the loss decreases over the epochs. The `test` function shows info about the encoding and decoding process:

```

('input', [[8, 4, 6, 2]])
('compressed', array([[ 0.78238308]], dtype=float32))
('reconstructed', array([[ 6.87756062,  2.79838109,  6.25144577,  2.23120356]],
dtype=float32))

```

Notice how we were able to compress a 4-dimensional vector into just 1 dimension and then decode it back into a 4-dimensional vector with some loss in data.

7.3 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 a lower likelihood of successfully converging. It's a double-edged sword. Go wield it in listing 7.6. We'll use that helper function later.

Listing 7.6 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 7.3. The batch version is shown in listing 7.7. 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 7.7 Batch learning

```
def train(self, data, batch_size=10):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for i in range(self.epoch):
            for j in range(500): //#A
                batch_data = get_batch(data, self.batch_size) //#B
                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')
```

#A Loop through various batch selections

#B Run the optimizer on a randomly selected batch

7.4 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 7.9.

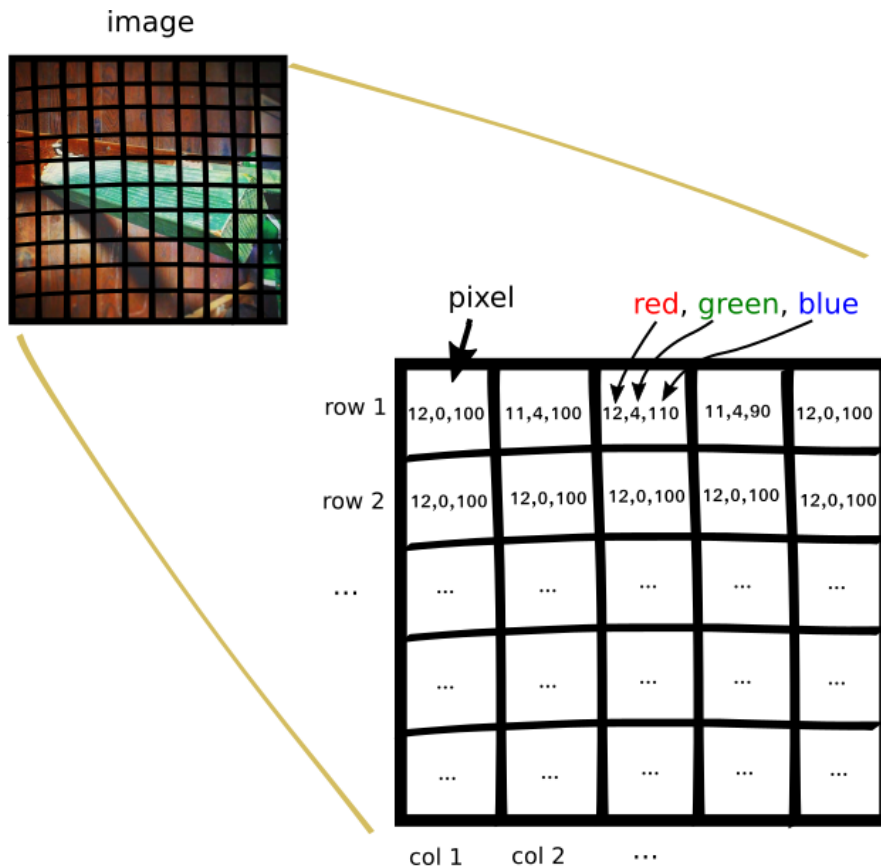


Figure 7.9 A colored image is composed of pixels, and each pixel contains a red, green, and blue value.

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*. Row-major order stores an array as a longer, single dimension set where we just put all the dimensions of an array onto the end of the first dimension, and is well supported by NumPy. This allows us to 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 7.10.

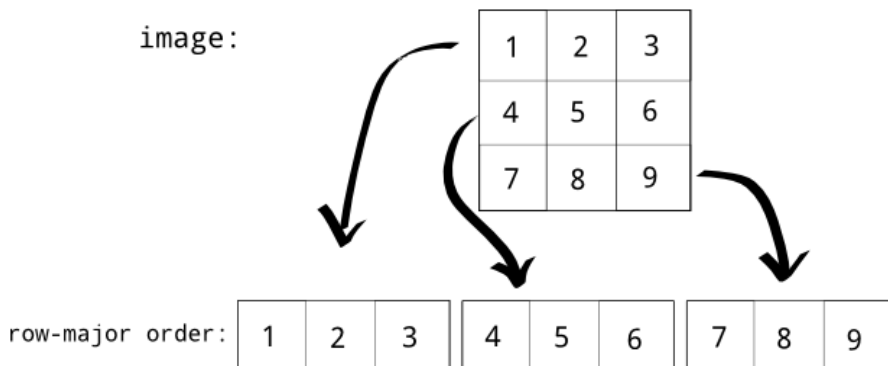


Figure 7.10 An image can be represented in row-major order. That way, we can represent a 2-dimensional structure as a 1-dimensional structure.

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 7.8 shows you how to load an image as in grayscale, resize it, and represent it in row-major order.

Listing 7.8 Loading images

```
from scipy.misc import imread, imresize

gray_image = imread(filepath, True) //#A
small_gray_image = imresize(gray_image, 1. / 8.) //#B
x = small_gray_image.flatten() //#C
```

#A load an image as grayscale
 #B resize it to something smaller
 #C 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.

EXERCISE 7.3 Can you name other image datasets online? Search online, and look around for more!

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 7.9 is provided from the CIFAR webpage. In a new file, called `main_imgs.py`, follow along to the listings.

Listing 7.9 Reading from extracted CIFAR-10 dataset

```
import pickle
```

```
def unpickle(file):    ##A
    fo = open(file, 'rb')
    dict = pickle.load(fo, encoding='latin1')
    fo.close()
    return dict
```

#A 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 7.10 shows how to loop through all the files and append the datasets to memory.

Listing 7.10 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):    ##A
    filename = './cifar-10-batches-py/data_batch_' + str(i)
    batch_data = unpickle(filename)    ##B
    if len(data) > 0:
        data = np.vstack((data, batch_data['data']))    ##C
        labels = np.hstack((labels, batch_data['labels']))    ##D
    else:
        data = batch_data['data']
        labels = batch_data['labels']
```

#A Loop through the 6 files

#B Load the file to obtain a Python dictionary

#C The rows of a data sample represent each sample, so we stack it vertically

#D Labels are simply 1-dimensional, so we'll stack them horizontally

Each image is represented as a series of red pixels, followed by green pixels, and then blue pixels. Listing 7.11 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 7.11 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 7.12.

Listing 7.12

```
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)
```

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. Beware, running this code may take up to 10 minutes. The output will trace loss values of every 10 epochs:

```
epoch 0: loss = 99.8635025024
epoch 10: loss = 35.3869667053
epoch 20: loss = 15.9411172867
epoch 30: loss = 7.66391372681
epoch 40: loss = 1.39575612545
epoch 50: loss = 0.00389165547676
epoch 60: loss = 0.00203850422986
epoch 70: loss = 0.00186171964742
epoch 80: loss = 0.00231492402963
epoch 90: loss = 0.00166488380637
epoch 100: loss = 0.00172081717756
epoch 110: loss = 0.0018497039564
epoch 120: loss = 0.00220602494664
epoch 130: loss = 0.00179589167237
epoch 140: loss = 0.00122790911701
epoch 150: loss = 0.0027100709267
epoch 160: loss = 0.00213225837797
epoch 170: loss = 0.00215123943053
epoch 180: loss = 0.00148373935372
epoch 190: loss = 0.00171591725666
```

See the book's GitHub repo for a full example of the output:

https://github.com/BinRoot/TensorFlow-Book/blob/master/ch07_autoencoder/Concept02_autoencoder_with_imgs.ipynb

7.5 Application of 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.

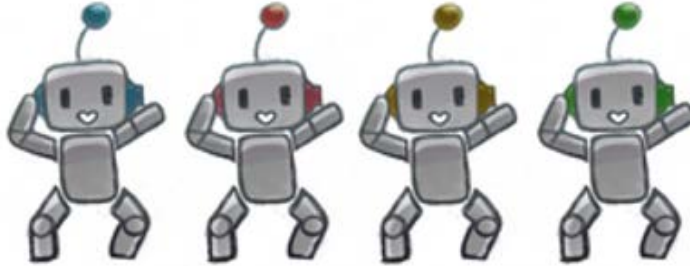
- A *stacked autoencoder* starts the same way a normal autoencoder does. It learns the encoding for an input into a smaller hidden layer by minimizing the reconstruction error. The hidden layer is now treated as the input to a new autoencoder that tries to encode the first layer of hidden neurons to 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 used 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.

7.6 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 flattening and grayscaling.

8

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 good 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 8.1 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*.

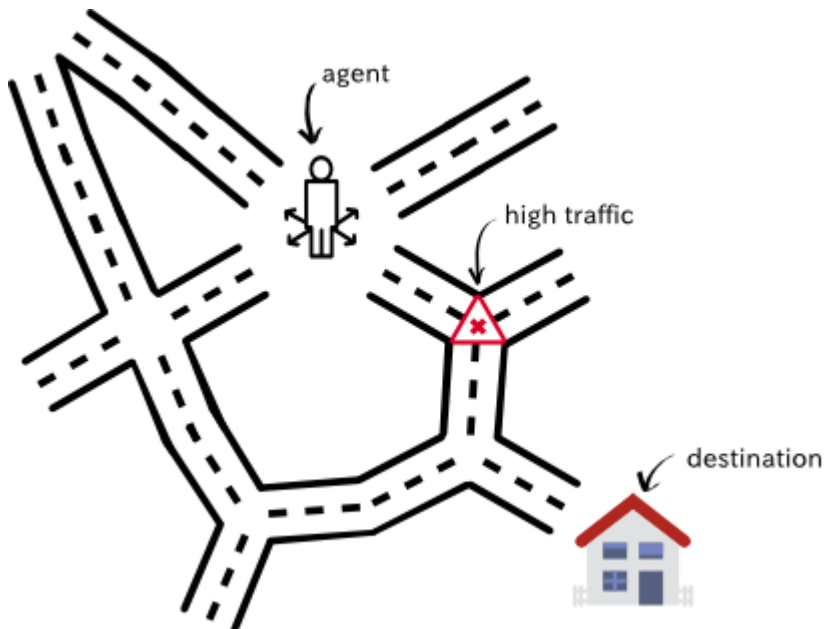


Figure 8.1 A person navigating his or her way to reach a destination in midst of traffic and unexpected situations is a problem set up for reinforcement learning.

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 expected rewards?

8.1 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 a good sequence of actions to take that maximize expected rewards.

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 8.2 demonstrates the interplay between states, actions, and rewards.

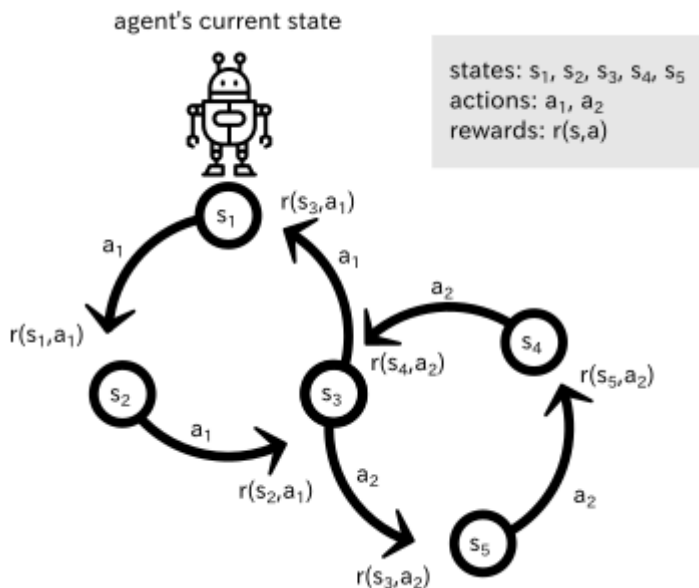


Figure 8.2 Actions are represented by arrows, and states are represented by circles. Performing an action on a state produces a reward. If you start at state s_1 , you can perform action a_1 to obtain a reward $r(s_1, a_1)$

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 on 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.

Most importantly, the actions we make aren't characterized solely by our observations at each moment. Otherwise, we're no smarter than bacteria, that act deterministically given their environment. There seems to be a lot more going on, and a simple RL model might not fully explain human behavior.

8.1.1 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*.

The goal of reinforcement learning is to discover a good policy. One of the most common ways to create that policy 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. Think of playing chess. Grabbing that queen might be really satisfying, but not if your opponent is guaranteed to put you in checkmate in the next five moves.

The best possible policy is called the *optimal policy*, and it's often the holy grail of reinforcement learning. Learning the optimal policy, as shown in figure 8.3, tells you the optimal action given any state.

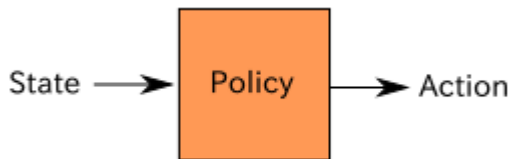


Figure 8.3 A policy suggests which action to take given a state.

Limitations of (Markovian) Reinforcement Learning

Most RL formulations assume that the best action to take can be figured out from simply knowing the current state, instead of considering the longer-term history of states and actions that got you there. This approach of making decisions based on the current state is called Markovian, and the general framework is often referred to as the Markov Decision Process (MDP).

Such situations where the state sufficiently captures what to do next can be modeled with RL algorithms in this chapter. However, most real-world situations are not Markovian, and therefore need a more realistic approach, such as a hierarchical representation of states and actions. In a grossly over-simplifying sense, hierarchical models are like context-free grammars, whereas MDPs are like finite state machines. The expressive leap of modeling a problem as an MDP to something more hierarchical can dramatically improve the optimality of the planning algorithm.

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 both the random and greedy policies.

8.1.2 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 using 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*, is shown in figure 8.4.

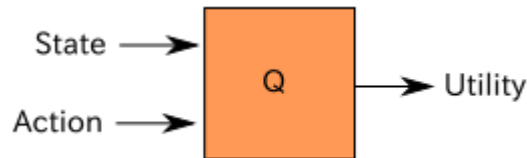


Figure 8.4 A utility function “Q” predicts the expected immediate reward plus rewards following an optimal policy given the state-action input.

EXERCISE 8.1 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 $r(s, a)$:

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a')$$

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. We can read the formula as “the value of this action is the immediate reward provided by taking this action, added to the discount factor times the best thing that can happen after that.”

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 $\alpha = 1$, both equations are identical).

$$Q(s, a) \rightarrow Q(s, a) + \alpha (r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Reinforcement learning can be solved if we know this $Q(s, a)$ function (called *Q-function*). Conveniently for us, *neural networks* (chapter 7) 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.

8.2 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 (cash on hand), and each action generates a reward (or loss).

EXERCISE 8.2 What are some possible disadvantages of using reinforcement learning for buying and selling stocks?

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 8.5 demonstrates one possible policy given stock market data.

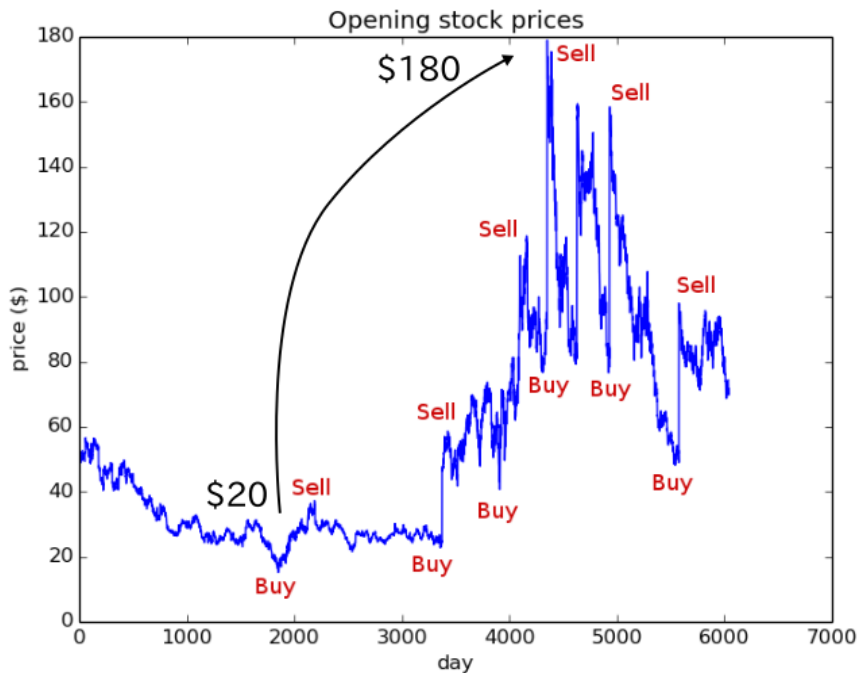


Figure 8.5 Ideally our algorithm should buy low and sell high. Doing so just once as shown in the figure might yield a reward of around \$160. But the real profit rolls in when you buy and sell more frequently. Ever heard the term high-frequency trading? It's about buying low and selling high as frequently 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!

8.3 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 8.1 Import relevant libraries

```
from yahoo_finance import Share // #A
from matplotlib import pyplot as plt // #B
import numpy as np // #C
import tensorflow as tf // #C
```

```
import random
```

```
#A For obtaining stock price raw data
#B For plotting stock prices
#C 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 8.2 for how to use the library and cache the data.

Listing 8.2 Helper function to get prices

```
def get_prices(share_symbol, start_date, end_date,
               cache_filename='stock_prices.npy'):
    try:    ##A
        stock_prices = np.load(cache_filename)
    except IOError:
        share = Share(share_symbol)    ##B
        stock_hist = share.get_historical(start_date, end_date)
        stock_prices = [stock_price['Open'] for stock_price in stock_hist]    ##C
        np.save(cache_filename, stock_prices)    ##D

    return stock_prices.astype(float)
```

```
#A Try to load the data from file if it has already been computed
#B Retrieve stock prices from the library
#C Extract only relevant info from the raw data
#D 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 8.3 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')
    plt.show()
```

We can grab some data and visualize it using the following snippet of code in listing 8.4.

Listing 8.4 Get data and visualize it

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

```
plot_prices(prices)
```

Figure 8.4 shows the produced figure from running listing 8.6.

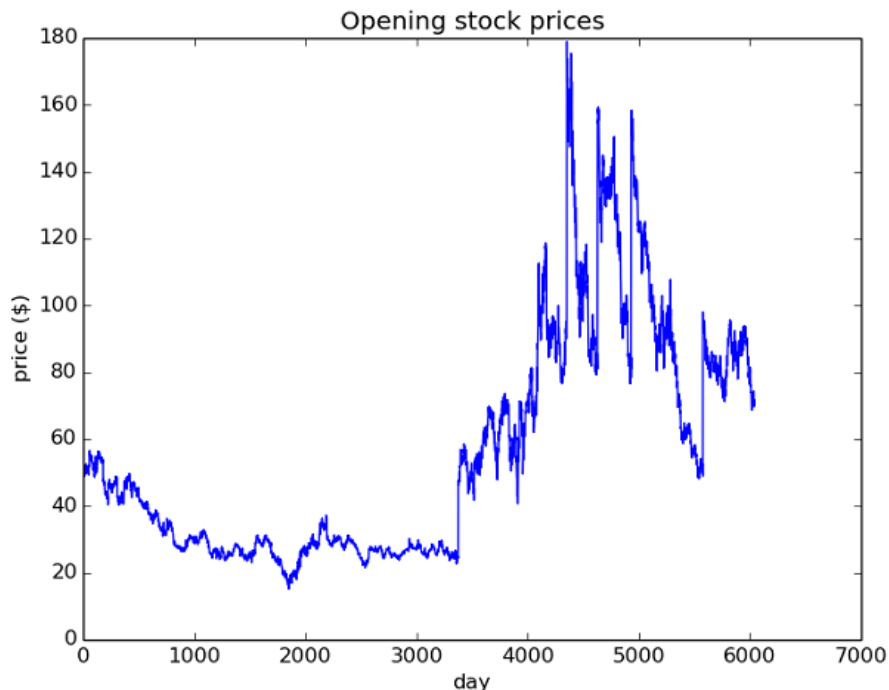


Figure 8.6 This chart summarizes the opening stock prices of Microsoft (MSFT) from 7/22/1992 to 7/22/2016. Wouldn't it have been nice to buy around day 3000 and sell around day 5000? Let's see if our code can learn to buy, sell, and hold to make the most money.

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 8.5 for an example and figure 8.7 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 8.5 Define a superclass for all decision policies

```
class DecisionPolicy:
    def select_action(self, current_state): // #A
        pass

    def update_q(self, state, action, reward, next_state): // #B
        pass
```

```
#A Given a state, the decision policy will calculate the next action to take
#B 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 8.7 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 8.6 for how to implement it.

Listing 8.6 Implement a random decision policy

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

    def select_action(self, current_state):    ##B
        action = random.choice(self.actions)
        return action
```

```
#A Inherit from DecisionPolicy to implement its functions
#B Randomly choose the next action
```

In listing 8.7, we assume a policy is given to us (such as the one from listing 8.6), and run it on the real-world stock-price data. This function takes care of exploration and exploitation at each interval of time. Figure 8.8 illustrates the algorithm from listing 8.7.

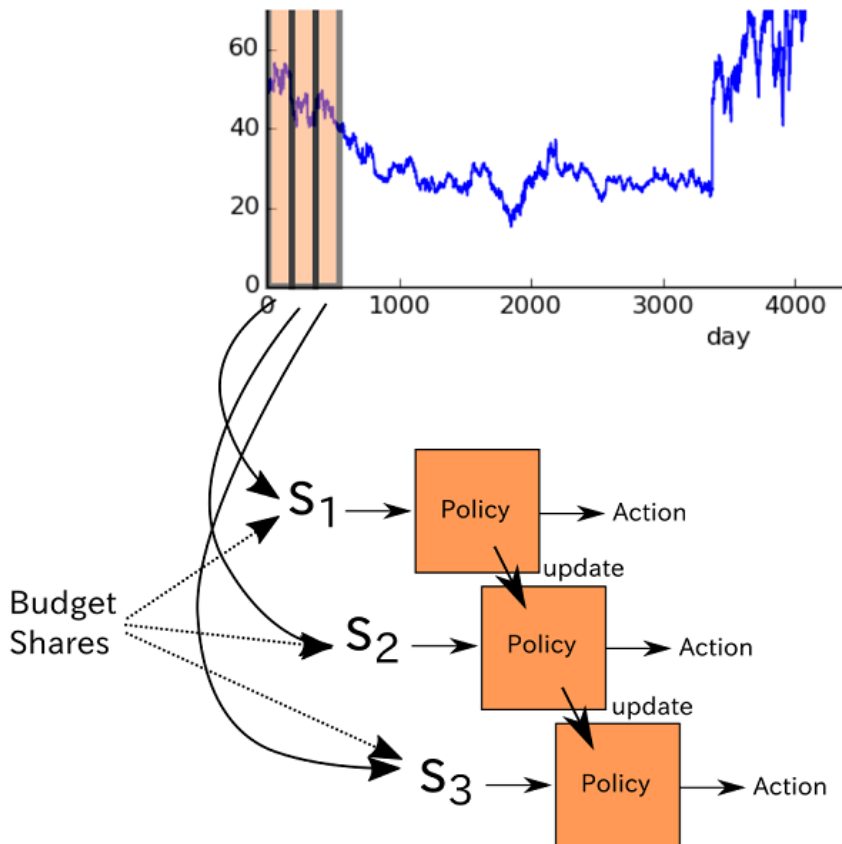


Figure 8.8 A rolling window of some size iterates through the stock prices as shown by the chart segmented to form states s_1 , s_2 , and s_3 . The policy suggests an action to take, which we may either choose to exploit or otherwise randomly explore another action. As we get rewards for performing an action, we can update the policy function over time.

Listing 8.7 Use a given policy to make decisions and return the performance

```
def run_simulation(policy, initial_budget, initial_num_stocks, prices, hist):
    budget = initial_budget // #A
    num_stocks = initial_num_stocks // #A
    share_value = 0 // #A
    transitions = list()
    for i in range(len(prices) - hist - 1):
        if i % 1000 == 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))) // #B
        current_portfolio = budget + num_stocks * share_value // #C
        action = policy.select_action(current_state, i) // #D
        share_value = float(prices[i + hist])
        if action == 'Buy' and budget >= share_value: // #E
```



```

        budget -= share_value
        num_stocks += 1
    elif action == 'Sell' and num_stocks > 0: ##E
        budget += share_value
        num_stocks -= 1
    else: ##E
        action = 'Hold'
    new_portfolio = budget + num_stocks * share_value ##F
    reward = new_portfolio - current_portfolio ##G
    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) ##H

portfolio = budget + num_stocks * share_value ##I
return portfolio

```

#A Initialize values that depend on computing the net worth of a portfolio
#B The state is a `hist+2`` dimensional vector. We'll force it to be a numpy matrix.
#C Calculate the portfolio value
#D Select an action from the current policy
#E Update portfolio values based on action
#F Compute new portfolio value after taking action
#G Compute the reward from taking an action at a state
#H Update the policy after experiencing a new action
#I 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 8.8 Run multiple simulations to calculate an average performance

```

def run_simulations(policy, budget, num_stocks, prices, hist):
    num_tries = 10 ##A
    final_portfolios = list() ##B
    for i in range(num_tries):
        final_portfolio = run_simulation(policy, budget, num_stocks, prices, hist) ##C
        final_portfolios.append(final_portfolio)
        print('Final portfolio: ${}'.format(final_portfolio))
    plt.title('Final Portfolio Value')
    plt.xlabel('Simulation #')
    plt.ylabel('Net worth')
    plt.plot(final_portfolios)
    plt.show()

```

#A Decide number of times to re-run the simulations
#B Store portfolio worth of each run in this array
#C Run this simulation
#D Average the values from all the runs

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

Listing 8.9 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'] //A
    hist = 3
    policy = RandomDecisionPolicy(actions) //B
    budget = 100000.0 //C
    num_stocks = 0 //D
    run_simulations(policy, budget, num_stocks, prices, hist) //E

```

#A Define the list of actions the agent can take
#B Initial a random decision policy
#C Set the initial amount of money available to use
#D Set the number of stocks already owned
#E 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 8.10 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 8.8.

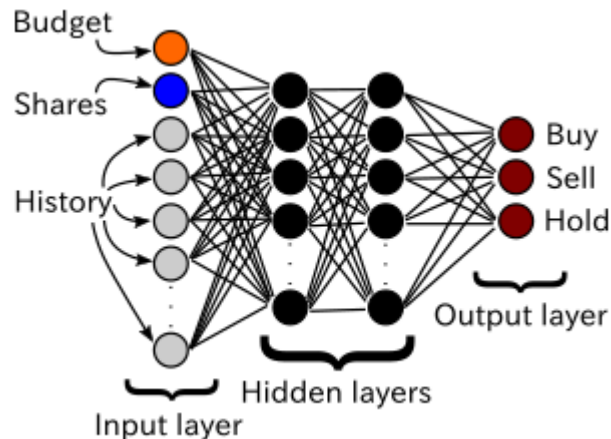


Figure 8.8 The input is the state space vector, with three outputs, one for each output's Q-value.

EXERCISE 8.3 What are other possible factors that our state-space representation ignores that can affect the stock prices? How could we factor them in to the simulation?

Listing 8.10 Implement a more intelligent decision policy

```

class QLearningDecisionPolicy(DecisionPolicy):

```

```

def __init__(self, actions, input_dim):
    self.epsilon = 0.95 ##A
    self.gamma = 0.3 ##A
    self.actions = actions
    output_dim = len(actions)
    h1_dim = 20 ##B

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

    loss = tf.square(self.y - self.q) ##F
    self.train_op = tf.train.AdagradOptimizer(0.01).minimize(loss) ##G
    self.sess = tf.Session() ##H
    self.sess.run(tf.global_variables_initializer()) ##H

def select_action(self, current_state, step):
    threshold = min(self.epsilon, step / 1000.)
    if random.random() < threshold: ##I

        # 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: ##J
        # 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): ##K
    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)
    current_action_idx = self.actions.index(action)
    action_q_vals[0, current_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})

```

#A Set the hyper-parameters from the Q-function

#B Set the number of hidden nodes in the neural networks

#C Define the input and output tensors

#D Design the neural network architecture

#E Define the op to compute the utility

#F Set the loss as the square error

#G Use an optimizer to update model parameters to minimize the loss

#H Set up the session and initialize variables

#I Exploit best option with probability epsilon

#J Explore random option with probability 1-epsilon

#K Update the Q-function by updating its model parameters

The resulting output when running the entire script is shown in figure 8.x.

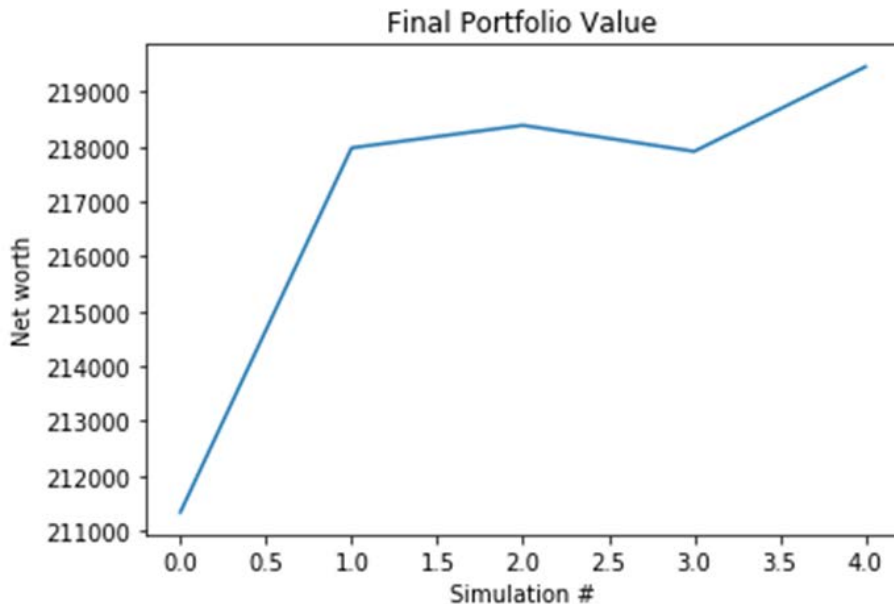


Figure 8.x The algorithm learns a good policy to trade Microsoft stocks

8.4 Applications of reinforcement learning

Reinforcement learning 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 the following 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 could beat top human Go players.
- Robotics and control: In March 2016, Google demonstrated a 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,

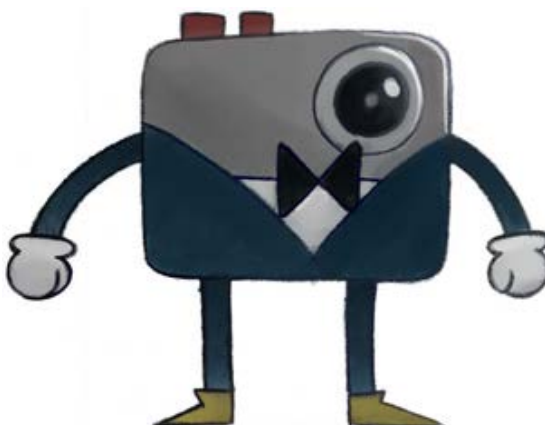
the robots were capable of grasping an object from the help of a camera input alone. To learn the simple concept of grasping an object, it required aggregating the knowledge of many robots spending many days to brute force attempts until enough patterns were detected. Clearly, there's a long way to go for generalizable learning of concepts, but it's an interesting start, nonetheless.

8.5 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.

9

Convolutional neural networks**This chapter covers**

- The components of a convolutional neural network
- How to classify natural images using deep learning
- Tips and tricks to improve neural network performance

Grocery shopping after an exhausting day is a taxing experience. My eyes get bombarded with too much information. Sales, coupons, colors, toddlers, flashing lights, crowded aisles are just a few examples of all the signals forwarded to my visual cortex, whether or not I actively try to pay attention. The visual system absorbs an abundance of information.

Ever heard the phrase “a picture is worth a thousand words?” At least that might be true for you or me, but can a machine find meaning within images as well? The photoreceptor cells in

our retinas pick up wavelengths of light, but that information doesn't seem to propagate up to our consciousness. After all, I can't put to words exactly what wavelengths of lights I'm picking up. Similarly, a camera picks up pixels, yet we want to squeeze out some form of higher-level knowledge instead, such as names or locations of objects. How do we get from pixels to human-level perception?

To achieve some intelligent meaning from raw sensory input with machine learning, we'll design a neural network model. In the previous chapters, we've seen a few types of neural networks models such as fully-connected ones (Chapter 8) or autoencoders (Chapter 7). In this chapter, we'll meet another type of model called a *convolutional neural network* (CNN), which performs exceptionally well on images and other sensory data such as audio. For example, a CNN model can reliably classify what object is being displayed in an image.

The CNN model that we'll implement in this chapter will learn how to classify images to 1 of 10 candidate categories. In effect, "a picture is worth only *one* word" out of just 10 possibilities. It's a tiny step toward human-level perception, but you have to start somewhere, right?

9.1 Drawback of neural networks

Machine learning is an eternal struggle of designing a model that is expressive enough to represent the data, yet not so flexible that it overfits and memorizes the patterns. Neural networks are proposed as a way to improve the expressive power, yet as you may guess, they often suffer from the pitfalls of overfitting.

REMINDER Overfitting is when your learned model performs exceptionally well on the training dataset, yet tends to perform poorly on the test dataset. The model is likely too flexible for what little data is available, and it ends up more or less memorizing the training data.

A quick and dirty heuristic you can use to compare the flexibility of two machine learning models is to count the number of parameters to be learned. As shown in figure 9.1, a fully connected neural network that takes in a 256x256 image and maps it to a layer of 10 neurons will have $256 * 256 * 10 = 655360$ parameters! Compare that to a model with perhaps only 5 parameters. It's very likely that the fully connected neural network can represent more complex data than the model with 5 parameters.

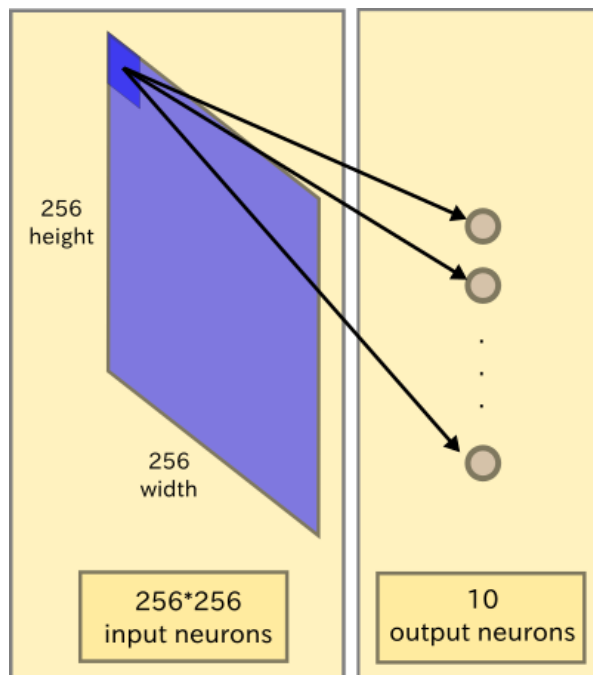


Figure 9.1 In a fully connected network, each pixel of an image is treated as an input. For a grayscale image of size 256×256 , that's 256×256 neurons! Connecting each neuron to 10 outputs yields $256 \times 256 \times 10 = 655360$ weights.

The next section introduces convolutional neural networks, which are a clever way to reduce the number of parameters. Instead of dealing with a fully connected network, the CNN approach ends up reusing the same parameter multiple times, as we'll see next.

9.2 Convolutional neural networks

The big idea behind convolutional neural networks is that a local understanding of an image is good enough. The practical benefit is that fewer parameters greatly improve the time it takes to learn as well as lowering the amount of data required to train the model.

Instead of a fully connected network of weights from each pixel, a CNN has just enough weights to look only at a small patch of the image. It's like reading a book using a magnifying glass; eventually you read the whole page, but you only look at a small patch of the page at any given time.

Consider a 256×256 image. Instead of your TensorFlow code processing the whole image at the same time, it can efficiently scan it chunk-by-chunk, say a 5×5 window. The 5×5 window slides along the image (usually left-to-right and top-to-bottom), as shown in figure 9.2. How "quickly" it slides is called its stride-length. For example, a stride-length of 2 means the sliding

window jumps around by 2 pixels. In TensorFlow, you can easily adjust the stride-length and window-size using the built-in library functions, as you'll soon see.

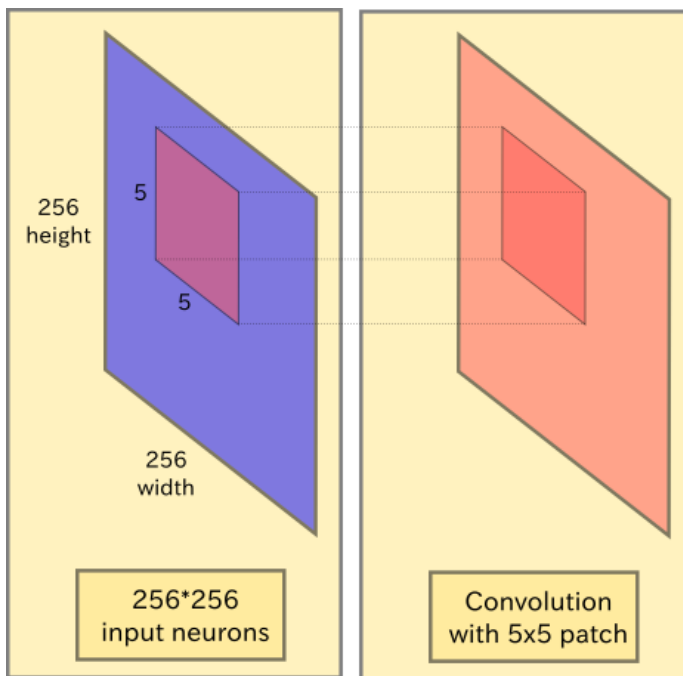


Figure 9.2 Convolving a 5x5 patch over an image as shown on the left produces another image, as shown on the right. In this case, the produced image is the same size as the original. Converting an original image to a convolved image requires only $5*5 = 25$ parameters!

This 5x5 window has an associated 5x5 matrix of weights.

DEFINITION A *convolution* is a weighted sum of the pixel-values of the image, as the window slides across the whole image. Turns out, this convolution process throughout an image with a weight matrix produces another image (of the same size, depending on the convention). By the way, *convolving* is the process of applying a convolution.

The sliding-window shenanigans happen in what is called the convolution layer of the neural network. A typical CNN has multiple convolution layers. Each convolutional layer typically generates many alternate convolutions, so the weight matrix is actually a tensor of $5*5*n$, where n is the number of convolutions.

As an example, let's say an image goes through a convolution layer on a weight matrix of $5*5*64$. That means it generates 64 convolutions by sliding a 5x5 window. Therefore, this

model has $5*5*64$ (=1,600) parameters, which is remarkably fewer parameters than a fully connected network, $256*256$ (=65,536)

The beauty of the CNN is that the number of parameters is independent of the size of the original image. We can run the same CNN on a 300x300 image, and the number of parameters will not change in the convolution layer!

9.3 Preparing the image

To get started with implementing CNNs in TensorFlow, let's first obtain some images to work with. The code listings in this section will help you set up a training dataset for the remainder of the chapter.

First, download the CIFAR-10 dataset from <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz>. This dataset contains 60,000 images, evenly split into 10 categories, which makes it a great resource for classification tasks. Go ahead and extract that file to your working directory. Figure 9.3 shows some examples of images from the dataset.



Figure 9.3 Here are a couple of examples of images from the CIFAR-10 dataset. Because they're only 32x32 in size, they're a bit difficult to see, but you can generally recognize some of the objects.

We've already used the CIFAR-10 dataset in the previous chapter about autoencoders, so let's pull up that code again. Listing 9.1 comes straight from the CIFAR-10 documentation located at <https://www.cs.toronto.edu/~kriz/cifar.html>. Place the code in a file called `cifar_tools.py`.

Listing 9.1 Loading images from a CIFAR-10 file in Python

```
import pickle

def unpickle(file):
    fo = open(file, 'rb')
    dict = pickle.load(fo, encoding='latin1')
    fo.close()
    return dict
```

Neural networks are already prone to overfitting, so it's essential that you do as much as you can to minimize that error. For that reason, always remember to clean the data before processing it.

Cleaning data is a core process in the machine learning pipeline. Listing 9.2 implements the following three steps for cleaning a dataset of images.

1. For example, if you have an image in color, try converting it to grayscale instead to lower the dimensionality of the input data, and consequently lower the number of parameters.
2. Also, consider center-cropping the image because maybe the edges of an image provide no useful information.
3. Last, but not least, don't forget to normalize your input by subtracting the mean and dividing by the standard deviation of each data sample so that the gradients during back-propagation don't change too dramatically. Listing 9.2 shows how you can clean a dataset of images using these techniques.

Listing 9.2 Cleaning data

```
import numpy as np

def clean(data):
    imgs = data.reshape(data.shape[0], 3, 32, 32) // #A
    grayscale_imgs = imgs.mean(1) // #B
    cropped_imgs = grayscale_imgs[:, 4:28, 4:28] // #C
    img_data = cropped_imgs.reshape(data.shape[0], -1)
    img_size = np.shape(img_data)[1]
    means = np.mean(img_data, axis=1)
    meansT = means.reshape(len(means), 1)
    stds = np.std(img_data, axis=1)
    stdsT = stds.reshape(len(stds), 1)
    adj_stds = np.maximum(stdsT, 1.0 / np.sqrt(img_size))
    normalized = (img_data - meansT) / adj_stds // #D
    return normalized
```

#A reorganize the data so it's a 32x32 matrix with 3 channels

```
#B Grayscale the image by averaging the color intensities
#C Crop the 32x32 image to a 24x24 image
#D Normalize the pixels' values by subtracting the mean and dividing by standard deviation
```

Let's collect all the images from CIFAR-10 into memory and run our cleaning function on them. Listing 9.3 sets up a convenient method to read, clean, and structure your data for use in TensorFlow. Include this in `cifar_tools.py` as well.

Listing 9.3 Preprocessing all CIFAR-10 files

```
def read_data(directory):
    names = unpickle('{} /batches.meta'.format(directory))['label_names']
    print('names', names)

    data, labels = [], []
    for i in range(1, 6):
        filename = '{} /data_batch_{}'.format(directory, i)
        batch_data = unpickle(filename)
        if len(data) > 0:
            data = np.vstack((data, batch_data['data']))
            labels = np.hstack((labels, batch_data['labels']))
        else:
            data = batch_data['data']
            labels = batch_data['labels']

    print(np.shape(data), np.shape(labels))

    data = clean(data)
    data = data.astype(np.float32)
    return names, data, labels
```

In another file called `using_cifar.py`, you can now use the method by importing `cifar_tools`. Listings 9.4 and 9.5 show how to sample a few images from the dataset and visualize them.

Listing 9.4 Using the `cifar_tools` helper function

```
import cifar_tools

names, data, labels = \
    cifar_tools.read_data('your/location/to/cifar-10-batches-py')
```

We can randomly select a few images and draw them along their corresponding label. Listing 9.5 does exactly that, so you can get a better understanding of the type of data you'll be dealing with.

Listing 9.5 Visualizing some images from the dataset

```
import numpy as np
import matplotlib.pyplot as plt
import random

def show_some_examples(names, data, labels):
```

```

plt.figure()
rows, cols = 4, 4 // #A
random_idx = random.sample(range(len(data)), rows * cols)
for i in range(rows * cols): // #B
    plt.subplot(rows, cols, i + 1)
    j = random_idx[i]
    plt.title(names[labels[j]])
    img = np.reshape(data[j, :], (24, 24))
    plt.imshow(img, cmap='Greys_r')
    plt.axis('off')
plt.tight_layout()
plt.savefig('cifar_examples.png')

show_some_examples(names, data, labels)

```

#A Change this to as many rows and columns as you desire

#B Randomly pick images from the dataset to show

By running the code in listing 9.5, you will generate a file called `cifar_examples.png` that will look similar to figure 9.3.

9.3.1 Generate filters

In this section, we'll convolve an image with a couple of random 5x5 patches, or also called *filter*. This is an important step in a convolutional neural network, so we'll carefully examine how data transforms. To understand a CNN model for image processing, it's wise to observe how an image filter transforms an image. Filters are a way to extract useful image features such as edges and shapes. With these features, we can train a machine learning model on them.

Remember: a feature vector is how we represent a data point. When we apply a filter to an image, the corresponding point in the transformed image is a feature – a feature that says “when you apply this filter to this point, is now has this new value.” The more filters we use on an image, the greater the dimension of the feature vector.

Open a new file called `conv_visuals.py`. Let's randomly initialize 32 filters. We will do so by defining a variable called `W` of size 5x5x1x32. The first two dimensions correspond to the filter size. The last dimension corresponds to the 32 different convolutions. The “1” in the variable's size corresponds to the input dimension, because the `conv2d` function is capable of convolving images of multiple channels. (In our example, we only care about grayscale images, so number of input channels is 1.)

Listing 9.6 provides the code to generate filters, which are shown in figure 9.4.



Figure 9.4 These are 32 randomly initialized matrices, each of size 5x5. They represent the filters we will use to convolve an input image.

Listing 9.6 Generate and visualize random filters

```
W = tf.Variable(tf.random_normal([5, 5, 1, 32]))  ##A

def show_weights(W, filename=None):
    plt.figure()
    rows, cols = 4, 8  ##B
    for i in range(np.shape(W)[3]):  ##C
        img = W[:, :, 0, i]
        plt.subplot(rows, cols, i + 1)
        plt.imshow(img, cmap='Greys_r', interpolation='none')
        plt.axis('off')
    if filename:
        plt.savefig(filename)
    else:
        plt.show()
```

##A Define the tensor representing the random filters

##B Define just enough rows and columns to show the 32 figures in Figure 9.4

##C Visualize each filter matrix

EXERCISE 9.1 Change listing 9.6 to generate 64 filters of size 3x3.

Use a session, as shown in listing 9.7, and initialize some weights using the `initialize_all_variables` op. Call the `show_weights` function to visualize random filters, as shown in figure 9.4.

Listing 9.7 Use a session to initialize weights

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    W_val = sess.run(W)
    show_weights(W_val, 'step0_weights.png')
```

9.3.2 Convolve using filters

The previous section prepared filters to use. In this section, we'll use TensorFlow's `convolve` function on our randomly generated filters. Listing 9.8 sets up some code to visualize the convolution outputs. We'll use it later just like the way we used `show_weights`.

Listing 9.8 Show convolution results

```
def show_conv_results(data, filename=None):
    plt.figure()
    rows, cols = 4, 8
    for i in range(np.shape(data)[3]):
        img = data[0, :, :, i] ##A
        plt.subplot(rows, cols, i + 1)
        plt.imshow(img, cmap='Greys_r', interpolation='none')
        plt.axis('off')
    if filename:
        plt.savefig(filename)
    else:
        plt.show()
```

##A Unlike listing 9.6, this time the shape of the tensor is different

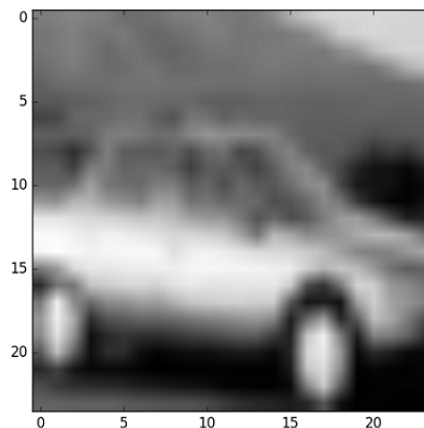


Figure 9.5 An example 24x24 image from the CIFAR-10 dataset.

Let's say we have an example input image, such as the one shown in figure 9.5. We can convolve the 24x24 image using 5x5 filters to produce many convolved images. All these convolutions are unique perspectives of looking at the same image. These different perspectives work together to comprehend what object exists in the image. In listing 9.9, we'll cover how to do this, step by step.

Listing 9.9 Visualizing convolution

```

//#A
raw_data = data[4, :]
raw_img = np.reshape(raw_data, (24, 24))
plt.figure()
plt.imshow(raw_img, cmap='Greys_r')
plt.savefig('input_image.png')

//#B
x = tf.reshape(raw_data, shape=[-1, 24, 24, 1])

//#C
b = tf.Variable(tf.random_normal([32]))
conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
conv_with_b = tf.nn.bias_add(conv, b)
conv_out = tf.nn.relu(conv_with_b)

//#D
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    conv_val = sess.run(conv)
    show_conv_results(conv_val, 'step1_convs.png')
    print(np.shape(conv_val))

conv_out_val = sess.run(conv_out)
show_conv_results(conv_out_val, 'step2_conv_outs.png')
print(np.shape(conv_out_val))

```

#A Get an image from the CIFAR dataset, and visualize it
#B Define the input tensor for the 24x24 image
#C Define the filters and corresponding parameters
#D Run the convolution on our selected image

Finally, by running the *conv2d* function in TensorFlow, we get the following 32 resulting images. The idea of convolving images is that each of the 32 convolutions capture different features about the image.

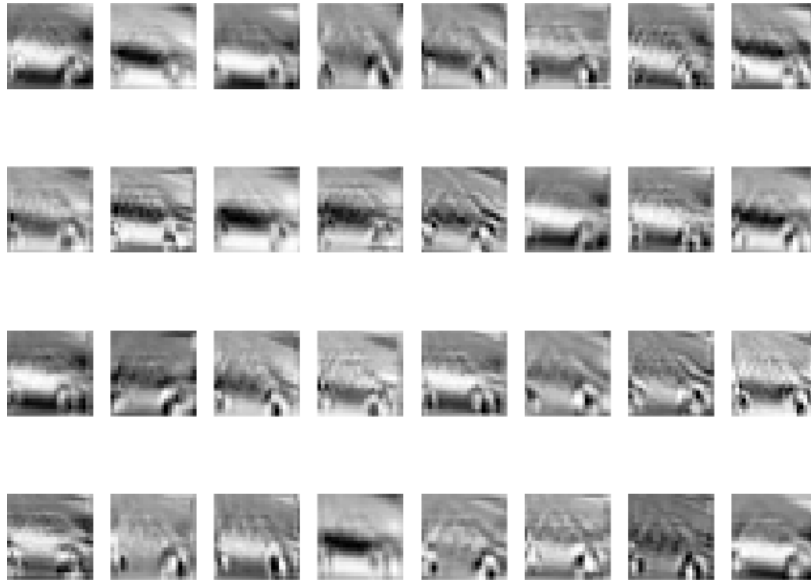


Figure 9.6 Resulting images from convolving the random filters on an image of a car.

By adding a bias term and an activation function such as *relu* (see listing 9.12 for an example), the convolution layer of the network behaves nonlinearly, which improves its expressiveness. See figure 9.7 for what each of the 32 convolution outputs become.

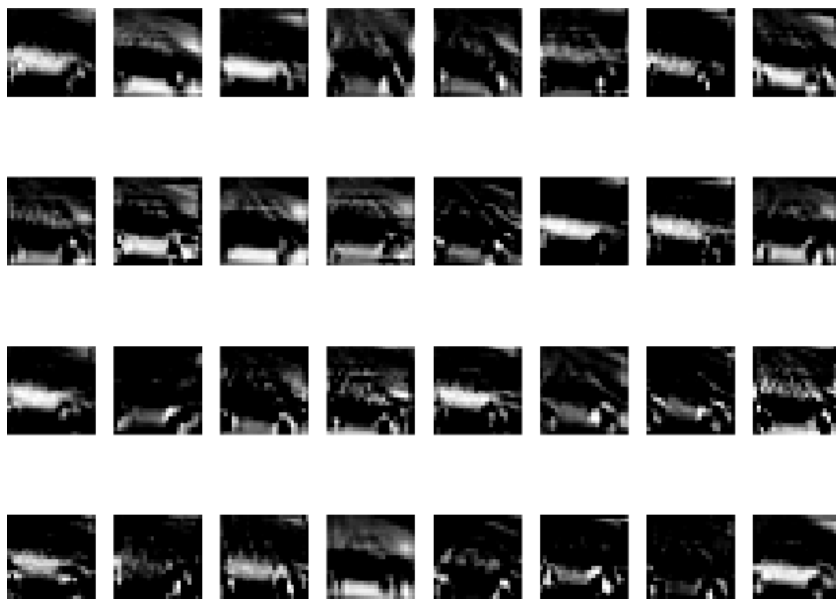


Figure 9.7 After adding a bias term and an activation function, the resulting convolutions can capture more powerful patterns within images.

9.3.3 Max-pooling

After a convolution layer extracts useful features, it's usually a good idea to reduce the size of the convolved outputs. Rescaling or subsampling a convolved output helps reduce the number of parameters, which in turn can help to not overfit the data.

This is the main idea behind a technique called *max-pooling*, which sweeps a window across an image and picks the pixel with the maximum value. Depending on the stride-length, the resulting image is a fraction of the size of the original. This is useful because it lessens the dimensionality of the data, consequently lowering the number of parameters in future steps.

EXERCISE 9.2 Let's say we want to max-pool over a 32x32 image. If the window size is 2x2, and the length of strides is 2, how big will the resulting max-pooled image be?

Place the following code in listing 9.10 within the Session context.

Listing 9.10 Running the maxpool function to subsample convolved images

```
k = 2
maxpool = tf.nn.max_pool(conv_out,
                        ksize=[1, k, k, 1],
                        strides=[1, k, k, 1],
                        padding='SAME')

with tf.Session() as sess:
    maxpool_val = sess.run(maxpool)
    show_conv_results(maxpool_val, 'step3_maxpool.png')
```

```
print(np.shape(maxpool_val))
```

As a result of running the code in listing 9.10, the max-pooling function halves the image size, and produces lower-resolution convolved outputs, as shown in figure 9.8.

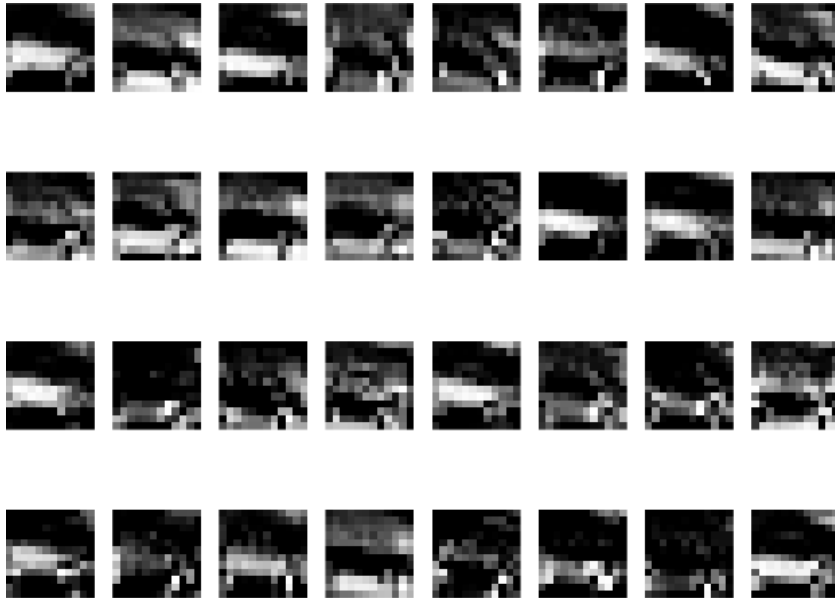


Figure 9.8 After running max-pool, the convolved outputs are halved in size, making the algorithm computationally faster without losing too much information.

We have the tools necessary to implement the full convolutional neural network. In the next section, we'll finally train the image classifier.

9.4 Implementing a convolutional neural network in TensorFlow

A convolutional neural network has multiple layers of convolutions and max-pooling. The convolution layer offers different perspectives on the image, while the maxpooling layer simplifies the computations by lowering the dimensionality without losing too much information.

Consider a full-size 256x256 image convolved by a 5x5 filter into 64 different convolutions. As shown in figure 9.9, each of the convolutions are subsampled using max-pooling to produce 64 smaller convolved images of size 128x128.

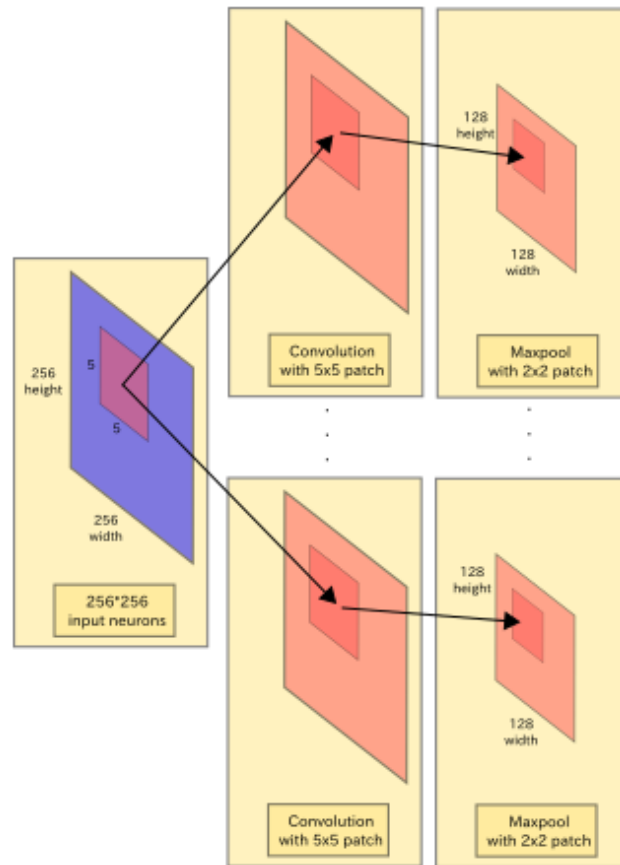


Figure 9.9 An input image (in blue) is convolved by multiple different 5x5 filters. The convolution layer includes an added bias term with an activation function, all together resulting in $5 \times 5 + 5 = 30$ parameters. Next, a maxpooling layer reduces the dimensionality of the data (which requires no extra parameters).

Now that we know how to make filters and use the convolution op, let's create a new source file. We'll start by defining all our variables. In listing 9.11, import all libraries, load the dataset, and finally define all variables.

Listing 9.11 Set up CNN weights

```
import numpy as np
import matplotlib.pyplot as plt
import cifar_tools
import tensorflow as tf

///<#A
names, data, labels = \
    cifar_tools.read_data('/home/binroot/res/cifar-10-batches-py')
```

```

//#B
x = tf.placeholder(tf.float32, [None, 24 * 24])
y = tf.placeholder(tf.float32, [None, len(names)])

//#C
W1 = tf.Variable(tf.random_normal([5, 5, 1, 64]))
b1 = tf.Variable(tf.random_normal([64]))

//#D
W2 = tf.Variable(tf.random_normal([5, 5, 64, 64]))
b2 = tf.Variable(tf.random_normal([64]))

//#E
W3 = tf.Variable(tf.random_normal([6*6*64, 1024]))
b3 = tf.Variable(tf.random_normal([1024]))

//#F
W_out = tf.Variable(tf.random_normal([1024, len(names)]))
b_out = tf.Variable(tf.random_normal([len(names)]))

#A Load the dataset
#B Define the input and output placeholders
#C Apply 64 convolutions of window-size 5x5
#D Then apply 64 more convolutions of window-size 5x5
#E Then we introduce a fully-connected layer
#F Lastly, define the variables for a fully-connected linear layer

```

In listing 9.12, we define a helper function to perform a convolution, add a bias term, and then an activation function. Together, these three steps form a convolution layer of the network.

Listing 9.12 Create a convolution layer

```

def conv_layer(x, W, b):
    conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
    conv_with_b = tf.nn.bias_add(conv, b)
    conv_out = tf.nn.relu(conv_with_b)
    return conv_out

```

Next, listing 9.13 shows how to define the max-pooling layer, by specifying the kernel and stride size.

Listing 9.13 Create a max-pool layer

```

def maxpool_layer(conv, k=2):
    return tf.nn.max_pool(conv, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME')

```

We can stack together the convolution and max-pool layers to define the convolutional neural network architecture. See listing 9.14 to define a possible CNN model. The last layer is typically just a fully-connected network connected to each of the 10 output neurons.

Listing 9.14 The full CNN model

```
def model():
    x_resaped = tf.reshape(x, shape=[-1, 24, 24, 1])

    ##A
    conv_out1 = conv_layer(x_resaped, W1, b1)
    maxpool_out1 = maxpool_layer(conv_out1)
    norm1 = tf.nn.lrn(maxpool_out1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75)

    ##B
    conv_out2 = conv_layer(norm1, W2, b2)
    norm2 = tf.nn.lrn(conv_out2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75)
    maxpool_out2 = maxpool_layer(norm2)

    ##C
    maxpool_resaped = tf.reshape(maxpool_out2, [-1, W3.get_shape().as_list()[0]])
    local = tf.add(tf.matmul(maxpool_resaped, W3), b3)
    local_out = tf.nn.relu(local)

    out = tf.add(tf.matmul(local_out, W_out), b_out)
    return out
```

#A Construct the first layer of convolution and maxpooling

#B Construct the second layer

#C Lastly, construct the concluding fully connected layers

9.4.1 Measuring performance

With a neural network architecture designed, the next step is to define a cost function that we wish to minimize. We'll use TensorFlow's function called `softmax_cross_entropy_with_logits`, which is best described by the official documentation (https://www.tensorflow.org/api_docs/python/tf/nn/softmax_cross_entropy_with_logits):

The function `softmax_cross_entropy_with_logits` measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

Because an image can belong to 1 of 10 possible labels, we will represent that choice as a 10-dimensional vector. All elements of this vector have a value 0, except the element corresponding to the label will have a value of 1. This representation, as we've seen in the earlier chapters, is called *one-hot encoding*.

As shown in listing 9.15, we'll calculate the cost via the cross-entropy loss function we mentioned in chapter 4. This returns the probability error for our classification. Note that this works only for simple classification – those where our classes are mutually exclusive, such as a truck can't also be a dog. There are many types of optimizers we can employ, but in this example, let's stick with the `AdamOptimizer`, which is a simple and fast optimizer (described in detail in https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer). It may be

worth playing around with the arguments to this in real-world applications, but it works well off the shelf.

Listing 9.15 Define ops to measure the cost and accuracy

```
model_op = model()

// #A
cost = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(logits=model_op, labels=y)
)

//#B
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(cost)

correct_pred = tf.equal(tf.argmax(model_op, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

#A Define the classification loss function

#B Define the training op to minimize the loss function

Finally, in the next section we will run the training op to minimize the cost of the neural network. Doing so multiple times throughout the dataset will learn the optimal weights (or parameters).

9.4.2 Training the classifier

In listing 9.16, we'll loop through the dataset of images in small batches to train the neural network. Over time, the weights will slowly converge to a local optimum to accurately predict the training images.

Listing 9.16 Train the neural network using CIFAR-10 dataset

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    onehot_labels = tf.one_hot(labels, len(names), on_value=1., off_value=0., axis=-1)
    onehot_vals = sess.run(onehot_labels)
    batch_size = len(data) // 200
    print('batch size', batch_size)
    for j in range(0, 1000): // #A
        print('EPOCH', j)
        for i in range(0, len(data), batch_size): // #B
            batch_data = data[i:i+batch_size, :]
            batch_onehot_vals = onehot_vals[i:i+batch_size, :]
            _, accuracy_val = sess.run([train_op, accuracy], feed_dict={x: batch_data, y:
            batch_onehot_vals})
            if i % 1000 == 0:
                print(i, accuracy_val)
        print('DONE WITH EPOCH')
```

#A Loop through 1000 epochs

#B Train the network in batches

That's it! We've successfully designed a convolutional neural network to classify images. Beware, it might take more than 10 minutes. If you're running this code on the CPU, it might even take hours! Could you imagine realizing a bug in your code after a day of waiting? That's why deep learning researchers use powerful computers and GPUs to speed up computations.

9.5 Tips and tricks to improve performance

The CNN we developed in this chapter is just a simple approach to solve the problem of image classification, but many techniques exist to improve the performance after you finish your first working prototype.

- **Augment data:** From a single image, you can easily generate new training images. As a start, just flip an image horizontally or vertically and you can quadruple your dataset size. You may also adjust the brightness of the image or the hue to ensure the neural network generalizes to other fluctuations. Lastly, you may even want to add random noise to the image to make the classifier robust to small occlusions. Scaling the image up or down can also be helpful; having exactly the same size items in your training images will almost guarantee you overfitting!
- **Early stopping:** Keep track of the training and testing error while you train the neural network. At first, both errors should slowly dwindle down, because the network is learning. But sometimes, the test error goes back up. This is a signal that the neural network has started overfitting on the training data, and is unable to generalize to previously unseen input. You should stop the training the moment you witness this phenomenon.
- **Regularize weights:** Another way to combat overfitting is by adding a regularization term to the cost function. We've already seen regularization in previous chapters, and the same concepts apply here.
- **Dropout:** TensorFlow comes with a handy function `tf.nn.dropout`, which can be applied to any layer of the network to reduce overfitting. It turns off a randomly selected number of neurons in that layer during training so that the network must be redundant and robust to inferring output.
- **Deeper:** A deeper architecture means adding more hidden layers to the neural network. If you have enough training data, it's been shown that adding more hidden layers improves performance.

EXERCISE 9.3 After the first iteration of a CNN architecture, try applying a couple tips and tricks mentioned in this chapter. Fine-tuning is, unfortunately, part of the process.

9.6 Application of convolutional neural networks

Convolutional neural networks blossom when the input contains sensor data from audio or images. Images, in particular, are of major interest in industry. For example, when you sign up for a social network, you usually upload a profile photo, not an audio recording of yourself

saying “hello.” It seems that humans are naturally more entertained by photos, so let’s see how CNNs can be used to detect faces in images.

The overall CNN architecture can be as simple or as complicated as your desire. You should start simple, and gradually tune your model until satisfied. There is no absolutely correct path, because facial recognition is still not completely solved. Researchers are still publishing papers that one-up the previous state-of-the-art solutions.

You should first obtain a dataset of images. One of the largest datasets of arbitrary images is called ImageNet (<http://image-net.org/>). Here, you can find negative examples for your binary classifier. To obtain positive examples of faces, you can find numerous datasets at the following sites that specialize in human faces:

- VGG Face Dataset
 - http://www.robots.ox.ac.uk/~vgg/data/vgg_face/
- Fddb: Face Detection Data Set and Benchmark
 - <http://vis-www.cs.umass.edu/fddb/>
- Face Detection and Pose Estimation
 - http://robotics.csie.ncku.edu.tw/Databases/FaceDetect_PoseEstimate.htm#Our_Database
- YouTube Faces Dataset
 - <https://www.cs.tau.ac.il/~wolf/ytfaces/>

9.7 Summary

The CNN we developed in this chapter is a simple approach to solving the problem of image classification. In this chapter, we covered one of the most difficult topics in machine learning, and used TensorFlow to accomplish a challenging classification problem.

- Convolutional neural networks make assumptions that capturing the local patterns of a signal are sufficient to characterize them, and thus reduce the number of parameters of a neural network.
- Cleaning data is vital to the performance of most machine learning models. The hour of time you spend to write code that cleans data is nothing compared to the amount of time it can take for a neural network to learn that cleaning function by itself.

10

Recurrent neural networks



This chapter covers

- The components of a recurrent neural network
- Designing a predictive model of timeseries data
- Using the timeseries predictor on real-world data

10.1 Contextual information

Back in school, I remember the sigh of relief when one of my midterm exams was made up of only true-or-false questions. I can't be the only one that assumed half the answers would be "true" and the other half would be "false."

I figured out answers to most of the questions, and left the rest to random guessing. Actually, I did something clever, a strategy that you might have employed as well. After counting my number of "true" answers, I realized a disproportionate amount of "false" answers were lacking. So, a majority of my guesses were "false" to balance the distribution.

It worked. I sure felt sly in the moment. What exactly is this feeling of craftiness that makes us feel so confident in our decisions, and how can we give a neural network the same power?

One answer to this question is to use context to answer questions. Contextual cues are important signals that can also improve the performance of machine learning algorithms. For example, imagine you want to examine an English sentence and tag the part of speech of each word.

The naive approach is to individually classify each word as a "noun," "adjective," and so on, without acknowledging its neighboring words. Consider trying that technique on the words in *this* sentence. The word "trying" was used as a verb, but depending on the context, you can also use it as an adjective, making parts-of-speech tagging a very *trying* problem.

A better approach would consider the context. To bestow neural networks with contextual cues, we'll study an architecture called a *recurrent neural network*. Instead of natural language data, we'll be dealing with continuous timeseries data, such as the stock-market prices we covered in previous chapters. By the end of the chapter, you'll be able to model the patterns in timeseries data to make predictions about future values.

10.2 Introduction to recurrent neural networks

To understand recurrent neural networks, let's first look at a simple architecture shown in figure 10.1. It takes as input a vector $X(t)$ and generates an output a vector $Y(t)$, at some time (t). The circle in the middle represents the hidden layer of the network.



Figure 10.1 A neural network with the input and output layer labeled as $X(k)$ and $Y(k)$, respectively

With enough input/output examples, you can learn the parameters of the network in TensorFlow. For instance, let's refer to the input weights as a matrix W_{in} , and the output weights as a matrix W_{out} . Assume there's just one hidden layer, referred to as a vector $Z(t)$.

As shown in figure 10.2, the first half of the neural network is characterized by the function $Z(t) = X(t) * W_{in}$, and the second half of the neural network takes the form $Y(t) = Z(t) * W_{out}$. Equivalently, if you prefer, the whole neural network is just the function $Y(t) = (X(t) * W_{in}) * W_{out}$.

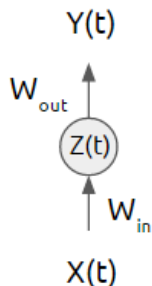


Figure 10.2 The hidden layer of a neural network can be thought of as a hidden representation of the data, which is encoded by the input weights and decoded by the output weights.

After spending nights fine-tuning the network, you probably want to start using your learned model in a real-world scenario. Typically, that implies you'll be calling the model multiple times, maybe even repeatedly one after another, as visualized in figure 10.3.

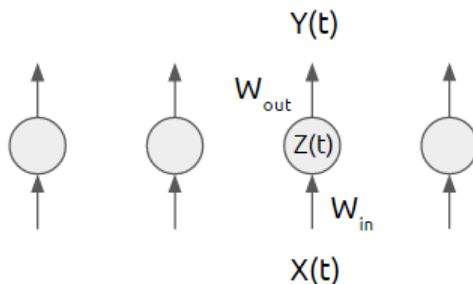


Figure 10.3 Often we end up running the same neural network multiple times, without using knowledge about the hidden states of the previous runs.

At each time t , when calling the learned model, this architecture does not take into account knowledge about the previous runs. It's like predicting stock-market trends by only looking at data from the current day. A better idea would be to exploit overarching patterns from a week's worth or months' worth of data.

A recurrent neural network (RNN) is different from a traditional neural network because it introduces a transition weight W to transfer information between time. Figure 10.4 shows the three weight matrices that must be learned in a RNN. The introduction of the transition weight means that the next state is now dependent on the previous model, as well as the previous state. This means our model now has a “memory” of what else we did!

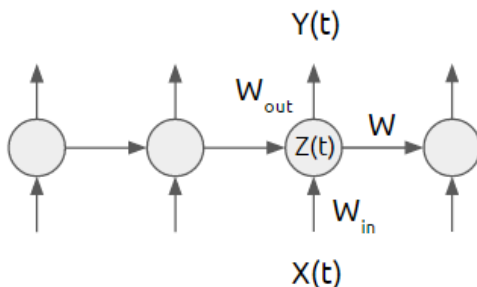


Figure 10.4 A recurrent neural network architecture can use the previous states of the network to its advantage.

Diagrams are nice, but you’re here to get your hands dirty. So, let’s get right to it! The next section shows how to use TensorFlow’s built-in RNN models. Then, we’ll use this RNN on real world timeseries data to predict the future!

10.3 Implementing a recurrent neural network

As we implement the RNN, we’ll use TensorFlow to do much of the heavy lifting. You won’t need to manually build up a network as shown earlier in figure 10.4, because the TensorFlow library already supports some robust RNN models.

REFERENCE For TensorFlow library information on RNN, please see <https://www.tensorflow.org/tutorials/recurrent>

One type of RNN model is called Long Short-Term Memory (LSTM). I admit, it’s a fun name. It means exactly what it sounds like, too: short-term patterns aren’t forgotten in the long-term.

The precise implementation detail of LSTM is not in the scope of this book. Trust me, a thorough inspection of the LSTM model would distract from the chapter, because there’s no definite standard yet. That’s where TensorFlow comes in to the rescue. It takes care of how the model is defined so you can use it out-of-the-box. It also means that as TensorFlow is updated in the future, we’ll be able to take advantage of improvements to the LSTM model without modifying our code.

FURTHER READING For understanding how to implement LSTM from scratch, I suggest the following explanation: <https://apaszke.github.io/lstm-explained.html>. The paper that describes the implementation of regularization we use in the listings below is available at <http://arxiv.org/abs/1409.2329>.

Let's begin by writing our code in a new file, called `simple_regression.py`. Import the relevant libraries, as shown in listing 10.1.

Listing 10.1 Import relevant libraries

```
import numpy as np
import tensorflow as tf
from tensorflow.contrib import rnn
```

Now, define a class called `SeriesPredictor`. The constructor, as shown in listing 10.2, will set up model hyper-parameters, weights, and the cost function.

Listing 10.2 Define a class and its constructor

```
class SeriesPredictor:
    def __init__(self, input_dim, seq_size, hidden_dim=10):

        self.input_dim = input_dim // #A
        self.seq_size = seq_size // #A
        self.hidden_dim = hidden_dim // #A

        self.W_out = tf.Variable(tf.random_normal([hidden_dim, 1]), name='W_out') // #B
        self.b_out = tf.Variable(tf.random_normal([1]), name='b_out') // #B
        self.x = tf.placeholder(tf.float32, [None, seq_size, input_dim]) // #B
        self.y = tf.placeholder(tf.float32, [None, seq_size]) // #B

        self.cost = tf.reduce_mean(tf.square(self.model() - self.y)) // #C
        self.train_op = tf.train.AdamOptimizer().minimize(self.cost) // #C

        self.saver = tf.train.Saver() // #D
```

#A Hyper-parameters

#B Weight variables and input placeholders

#C Cost optimizer

#D Auxiliary ops

Next, let's use TensorFlow's built-in RNN model called `BasicLSTMCell`. The hidden dimension of the cell passed into the `BasicLSTMCell` object is the dimension of the hidden state that gets passed through time. We can run this cell with some data using the `rnn.dynamic_rnn` function, to retrieve the output results. Listing 10.3 details how to use TensorFlow to implement a predictive model using LSTM.

Listing 10.3 Define the RNN model

```
def model(self):
    """
    :param x: inputs of size [T, batch_size, input_size]
```

```

:param W: matrix of fully-connected output layer weights
:param b: vector of fully-connected output layer biases
"""
cell = rnn.BasicLSTMCell(self.hidden_dim) #A
outputs, states = tf.nn.dynamic_rnn(cell, self.x, dtype=tf.float32) #B
num_examples = tf.shape(self.x)[0]
W_repeated = tf.tile(tf.expand_dims(self.W_out, 0), [num_examples, 1, 1])
#C
out = tf.matmul(outputs, W_repeated) + self.b_out
out = tf.squeeze(out)
return out

```

#A Create a LSTM cell

#B Run the cell on the input to obtain tensors for outputs and states

#C Compute the output layer as a fully connected linear function

With a model and cost-function defined, we can now implement the training function, which will learn the LSTM weights given example input/output pairs. As listing 10.4 shows, you open a session and repeatedly run the optimizer on the training data.

BY THE WAY You can use cross-validation to figure out how many iterations to train the model. In our case here, we assume a fixed number of epochs. Some good insights and answers can be found through online Q&A sites such as the following:
https://www.researchgate.net/post/How_does_one_choose_optimal_number_of_epochs.

After training, save the model to file so we can load it later.

Listing 10.4 Train the model on a dataset

```

def train(self, train_x, train_y):
    with tf.Session() as sess:
        tf.get_variable_scope().reuse_variables()
        sess.run(tf.global_variables_initializer())
        for i in range(1000): #A
            _, mse = sess.run([self.train_op, self.cost], feed_dict={self.x: train_x,
self.y: train_y})
            if i % 100 == 0:
                print(i, mse)
        save_path = self.saver.save(sess, 'model.ckpt')
        print('Model saved to {}'.format(save_path))

```

#A Run the train op 1000 times

Let's say all went well, and our model has successfully learned parameters. Next, we'd like to evaluate the predictive model on other data. Listing 10.5 loads the saved model, and runs the model in a session by feeding in some test data. If a learned model doesn't perform well on testing data, then we can try tweaking the number of hidden dimensions of the LSTM cell.

Listing 10.5 Test the learned model

```

def test(self, test_x):

```

```

with tf.Session() as sess:
    tf.get_variable_scope().reuse_variables()
    self.saver.restore(sess, './model.ckpt')
    output = sess.run(self.model(), feed_dict={self.x: test_x})
    print(output)

```

It's done! But just to convince ourselves that it works, let's make up some data and try to train the predictive model. In listing 10.6, we'll create some input sequences, called `train_x`, and some corresponding output sequences, called `train_y`.

Listing 10.6 Train and test on some dummy data

```

if __name__ == '__main__':
    predictor = SeriesPredictor(input_dim=1, seq_size=4, hidden_dim=10)
    train_x = [[1], [2], [5], [6]],
              [[5], [7], [7], [8]],
              [[3], [4], [5], [7]]]
    train_y = [[1, 3, 7, 11],
              [5, 12, 14, 15],
              [3, 7, 9, 12]]
    predictor.train(train_x, train_y)

    test_x = [[1], [2], [3], [4]], #A
              [[4], [5], [6], [7]]] #B
    predictor.test(test_x)

```

#A predicted result should be 1, 3, 5, 7

#B predicted result should be 4, 9, 11, 13

You can treat this predictive model as a black-box, and train it using real-world timeseries data for prediction. In the next section, we'll get some data to work with.

10.4 A predictive model for timeseries data

Timeseries data is abundantly available online. For this example, we'll use some data about international airline passengers for a specific period. You can obtain this data from <https://datamarket.com/data/set/22u3/international-airline-passengers-monthly-totals-in-thousands-jan-49-dec-60#lds=22u3&display=line>. Clicking that link will take you to a nice plot of the timeseries data, as shown in figure 10.5.

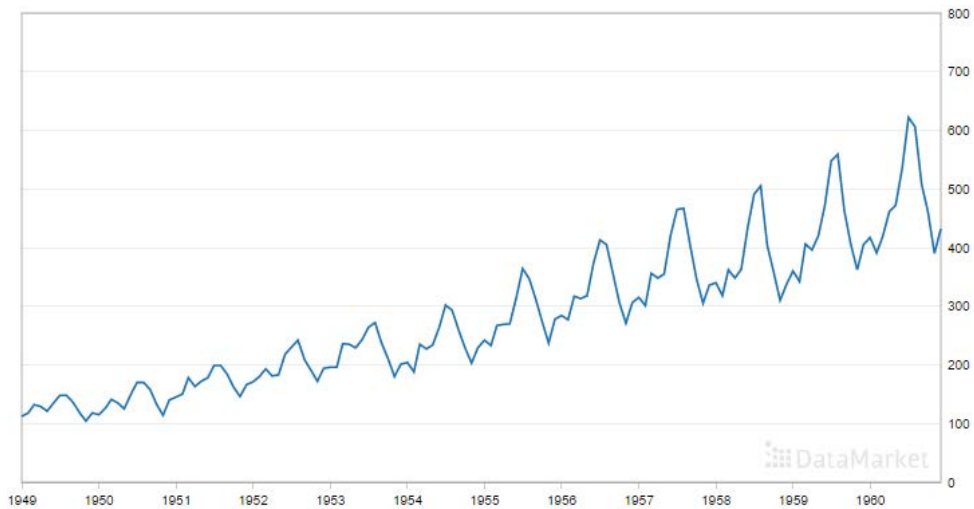


Figure 10.5 Raw data showing the number of international airline passengers throughout the years

You can download the data by choosing the *Export* tab and then selecting *CSV (,)* in the *Export group*. You'll have to manually edit the CSV file to remove the header line, as well as the additional footer line.

In a file called `data_loader.py`, add the code shown in listing 10.7.

Listing 10.7 Loading data

```
import csv
import numpy as np
import matplotlib.pyplot as plt

def load_series(filename, series_idx=1):
    try:
        with open(filename) as csvfile:
            csvreader = csv.reader(csvfile)
            ##A
            data = [float(row[series_idx]) for row in csvreader
                    if len(row) > 0]
            normalized_data = (data - np.mean(data)) / np.std(data) ##B
            return normalized_data
    except IOError:
        return None

def split_data(data, percent_train=0.80):
    num_rows = len(data) * percent_train ##C
    return data[:num_rows], data[num_rows:] ##D
```

#A Loop through the lines of the file and convert to a floating point number
 #B Pre-process the data by mean-centering and dividing by standard deviation
 #C Calculate training data samples

#D Split the dataset into training and testing

Here, we define two functions, `load_series` and `split_data`. The first function loads the timeseries file on disk, and normalizes it, and the other function divides the dataset into two components for training and testing.

Because we'll be evaluating the model multiple times to predict future values, let's modify the `test` function from the `SeriesPredictor`. It now takes as argument the session, instead of initializing the session on every call. See listing 10.8 for this tweak.

Listing 10.8 Modify the test function to pass in the session

```
def test(self, sess, test_x):
    tf.get_variable_scope().reuse_variables()
    self.saver.restore(sess, './model.ckpt')
    output = sess.run(self.model(), feed_dict={self.x: test_x})
    return output
```

We can now train the predictor by loading the data in the acceptable format. Listing 10.9 shows how to train the network, and then use the trained model to predict future values. We will generate the training data (`train_x` and `train_y`) to look like those shown previously in listing 10.6.

Listing 10.9

```
if __name__ == '__main__':
    seq_size = 5
    predictor = SeriesPredictor(
        input_dim=1, #A
        seq_size=seq_size, #B
        hidden_dim=100) #C

    #D
    data = data_loader.load_series('international-airline-passengers.csv')
    train_data, actual_vals = data_loader.split_data(data)

    train_x, train_y = [], []
    for i in range(len(train_data) - seq_size - 1): #E
        train_x.append(np.expand_dims(train_data[i:i+seq_size], axis=1).tolist())
        train_y.append(train_data[i+1:i+seq_size+1])

    test_x, test_y = [], [] #F
    for i in range(len(actual_vals) - seq_size - 1):
        test_x.append(np.expand_dims(actual_vals[i:i+seq_size], axis=1).tolist())
        test_y.append(actual_vals[i+1:i+seq_size+1])

    predictor.train(train_x, train_y, test_x, test_y) #G

    #H
    with tf.Session() as sess:
        predicted_vals = predictor.test(sess, test_x)[: ,0]
        print('predicted_vals', np.shape(predicted_vals))
        plot_results(train_data, predicted_vals, actual_vals, 'predictions.png')
```

```

prev_seq = train_x[-1]
predicted_vals = []
for i in range(20):
    next_seq = predictor.test(sess, [prev_seq])
    predicted_vals.append(next_seq[-1])
    prev_seq = np.vstack((prev_seq[1:], next_seq[-1]))
plot_results(train_data, predicted_vals, actual_vals, 'hallucinations.png')

```

#A The dimension of each element of the sequence is a scalar (1-dimensional)
#B Length of each sequence
#C Size of the RNN hidden dimension
#D Load the data
#E Slide a window through the time-series data to construct the training dataset
#F Do the same window sliding strategy to construct the test dataset
#G Train a model on the training dataset
#H Visualize the model's performance

The predictor will generate two graphs. The first is prediction results of the model given ground truth values, as shown in figure 10.5.

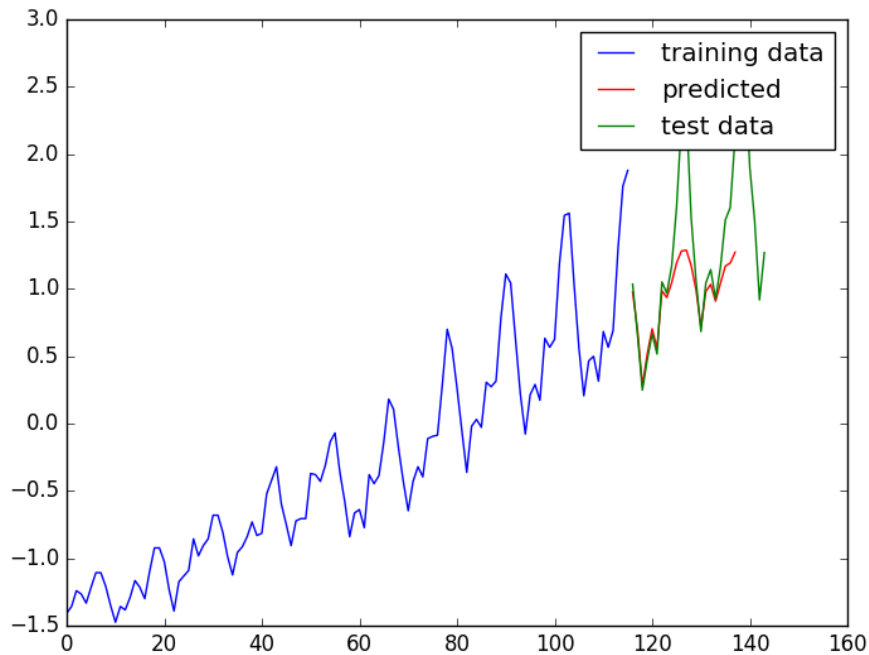


Figure 10.5 The predictions match trends fairly well when tested against ground-truth data.

The other graph shows the predictions results where only the training data was given (blue line) and nothing else. This procedure has less information available, but it still did a good job matching trends of the data.

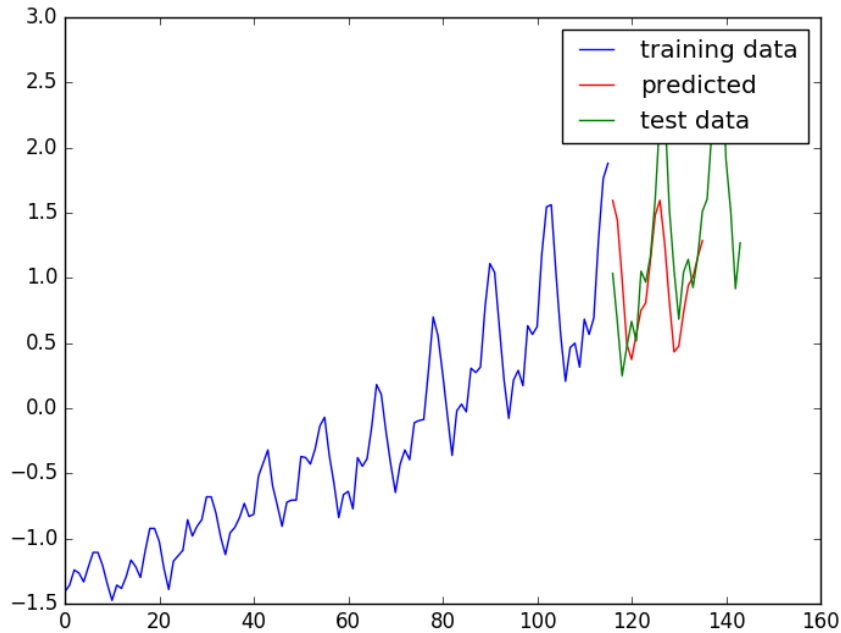


Figure 10.6 If the algorithm uses previously predicted results make further predictions, then general trend matches well, but not specific bumps.

We can use a timeseries predictor to reproduce realistic fluctuations in data. Imagine predicting market booms and bust cycles based on the tools you've learned so far. What are you waiting for? Grab some market data and learn your own predictive model!

10.5 Application of recurrent neural networks

Recurrent neural networks are meant to be used with sequential data. Because audio signals are a dimension lower than videos (linear signal versus two-dimensional pixel array), it's a lot easier to get started with audio time-series data. Consider how much speech recognition has improved over the years: it's becoming a tractable problem!

Like the audio histogram analysis we conducted in chapter 5 on clustering audio data, most speech recognition pre-processing involves representing the sound into a chromagram of sorts. Specifically, a common technique is to use a feature called Mel Frequency Cepstral Coefficients (MFCCs). A good introduction on the feature extraction is outlined in the following blog post: <http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/>.

Next, you'll need a dataset to train your model. A few popular ones include the following:

- LibriSpeech
 - <http://www.openslr.org/12/>

- TED-LIUM
 - <http://www.openslr.org/7/>
- VoxForge
 - <http://www.voxforge.org/>

An in-depth walkthrough of a simple speech-recognition implementation in TensorFlow using these datasets is available online: <https://svds.com/tensorflow-rnn-tutorial/>.

10.6 Summary

Recurrent neural networks, specifically LSTM models, are often a difficult topic to understand. Thanks to TensorFlow's abstraction, you can use the model without distracting yourself with the inner workings.

- A Recurrent Neural Network makes use of information from the past. That way, it can make predictions in data with high temporal dependencies.
- TensorFlow comes with Recurrent Neural Network models out-of-the-box.
- Timeseries prediction is a useful application for RNNs because of temporal dependencies in the data.

11

Sequence-to-sequence models for chatbots

This chapter covers

- **Seq-to-seq architecture**
- **Vector embedding of words**
- **Implementing a chat-bot using real-world data**

Talking to customer service over the phone is a burden for both the customer and the company. Service providers pay a good chunk of money to hire these customer service representatives, but what if it's possible to automate most of this effort? Can we develop software to interface with customers through natural language?

The idea is not as farfetched as you might think. Chatbots are getting a lot of hype due to unprecedented developments in natural language processing using deep-learning techniques. Perhaps, given enough training data, a chatbot could learn to navigate most-commonly addressed customer problems through natural conversations. If the chatbot were truly efficient, it could not only save the company money from needing to hire representatives, but even accelerate the customer's search for an answer.

In this chapter, we'll build a chatbot by feeding a neural network thousands of examples of input and output sentences. In this chapter, your training dataset is a pair of English utterances: for example, if you ask "how are you?", it should respond "fine, thank you".

BY THE WAY In this chapter, we're thinking of sequences and sentences are interchangeable concepts. In our implementation, a sentence will be a sequence of letters. Another common approach is to represent a sentence as a sequence of words.

In effect, the algorithm will try to produce an intelligent natural language response to each natural language query. We'll be implementing a neural network that uses two primary concepts taught in previous chapters: multi-class classification and recurrent neural networks (RNNs).

11.1.1 Classification

Remember, classification is a machine learning approach to predict the category of an input data item. Furthermore, multi-class classification allows for more than 2 classes. We've seen in Chapter 4 how exactly to implement such an algorithm in TensorFlow. Specifically, the cost function between the model's prediction (a sequence of numbers) and the ground truth (a one-hot vector) tries to find the distance between two sequences using the cross-entropy loss.

REMINDER A one-hot vector is like an all-zero vector, except one of the dimensions has a value of one.

In this case of implementing a chatbot, we'll use a variant of the cross-entropy loss to measure the difference between two sequences: the model's response (which is a sequence) against the ground truth (which is also a sequence).

EXERCISE 11.1 In TensorFlow, we can use the cross-entropy loss function to measure the similarity between a one-hot vector, such as (1, 0, 0), against a neural network's output, such as (2.34, 0.1, 0.3). On the other hand, English sentences are not numeric vectors. How can you use the cross-entropy loss to measure the similarity between English sentences?

11.1.2 Recurrent neural networks

You may recall that RNNs are a neural network design for incorporating not only input from the current time-step, but also state information from previous inputs. Chapter 10 covered these in great detail, and they'll be used again in this chapter. RNNs represent input and outputs as time-series data, which is exactly what we need to represent sequences.

A naive idea is to simply use an out-of-the-box RNN to implement a chatbot. Let's see why this is a bad approach. The input and output of the RNN are natural language sentences, so the inputs ($x_t, x_{t-1}, x_{t-2}, \dots$) and outputs ($y_t, y_{t-1}, y_{t-2}, \dots$) can be sequences of words. The problem in using an RNN to model conversations is that the RNN produces an output result immediately. So, if your input is a sequence of words such as ("How", "are", "you"), the first output word will depend on only the first input word. The output sequence item y_t of the RNN could not look ahead to future parts of the input sentence to make a decision; it would be limited by knowledge of only previous input sequences ($x_t, x_{t-1}, x_{t-2}, \dots$). Effectively, the naive RNN model tries to come up with a response to the user's query before they finished asking it, which can lead to incorrect results.

Instead, we'll end up using two RNNs: one for the input sentence, and the other for the output sequence. Once the input sequence is finished processing by the first RNN, it'll send the hidden state to the second RNN to process the output sentence. You can see the two RNNs labeled as "Encoder" and "Decoder" in figure 11.1.

Seq-to-seq model overview

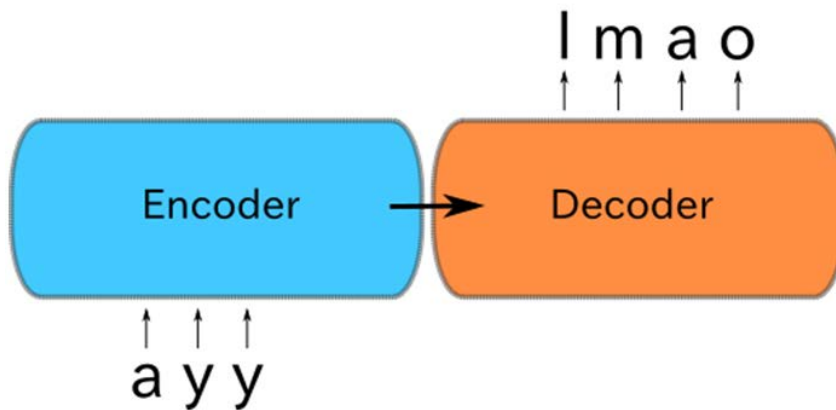


Figure 11.1 Here's a high level view of our neural network model. The input "ayy" is passed into the encoder RNN, and the decoder RNN is expected to respond with "lmao". These are just toy examples for our chat-bot, but you could imagine more complicated pairs of sentences for the input and output.

11.1.3 Classification and RNNs

We're bringing to bear concepts of multi-class classification and RNNs from previous chapters into designing a neural network that learns to map an input sequence to an output sequence. The RNNs provide a way of encoding the input sentence, passing a summarized state vector to the decoder, and then decoding it to a response sentence. To measure the cost between the model's response and the ground truth, we look to the function used in multi-class classification, the cross-entropy loss, for inspiration.

The name for this architecture is called a *sequence-to-sequence (seq-to-seq) neural network architecture*. The training data you use will be thousands of pairs of sentences mined from movie scripts. The algorithm will observe these dialogue examples, and eventually learn to form responses to arbitrary queries you might ask it.

POP QUIZ What other industries could benefit from a chatbot?

By the end of the chapter, we'll have our very own chatbot that can respond somewhat intelligently to our queries. Of course, it won't be perfect, because this model always responds the same way for the same input query.

Consider for example that you're travelling to a foreign country without any ability to speak their language. A clever salesman hands you a book, claiming it's all you need to respond to sentences in the foreign language. You're supposed to use it like a dictionary. When someone says a phrase in the foreign language, you can look it up, and the book will have the response written out for you to read aloud: "If someone says *Hello*, you say *Hi*".

Sure, it might be a practical loop-up table for small talk, but can a look-up table really get you the correct response for arbitrary dialogue? Of course not! Consider looking up the question “Are you hungry?”. The answer to that question is written out in the book and will never change.

The look-up table is missing state-information, which is a key component in dialogue. In our seq-to-seq model, we’ll suffer a similar issue; but, it’s a good start! Believe it or not, as of 2017, hierarchical state representation for intelligent dialogue is still not the norm; many chatbots start out with these seq-to-seq models.

11.2 Seq-to-seq architecture

The seq-to-seq model attempts to learn a neural network that predicts an output sequence from an input sequence. Sequences are a little different from traditional vectors, because a sequence implies an ordering of events.

Time is an intuitive way to order events: we usually end up alluding to words related to time, such as temporal, time-series, past, and future. For example, we like to say that RNNs propagate information to *future time-steps*. Or, RNNs capture *temporal dependencies*.

BY THE WAY RNNs are covered in detail in Chapter 10.

The *seq-to-seq model* is implemented using multiple RNNs. A single RNN cell is visualized in Figure 11.2, and it serves as the building block for the rest of the seq-to-seq model architecture.

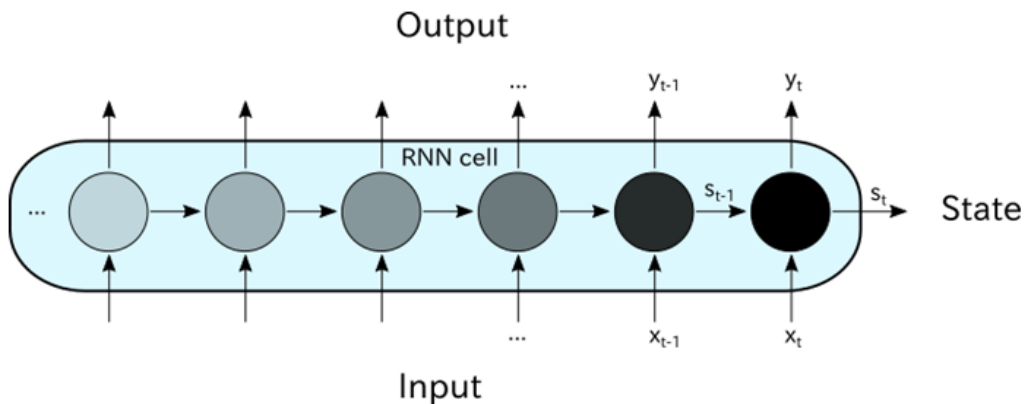


Figure 11.2 The input, output, and states of an RNN are outlined in this figure. We can ignore the intricacies of how exactly an RNN is implemented. All that matters is formatting of our input and output.

First, you’ll learn how to stack RNNs on top of each other to improve the model’s complexity. Then you’ll learn how to pipe the hidden state of one RNN to another RNN, so that

we can have an “encoder” and “decoder” network. As you’ll begin to see, it’s fairly easy to start using RNNs.

After that you’ll get an introduction to converting natural language sentences into a sequence of vectors. After all, RNNs only understand numeric data, so we’ll absolutely need this conversion process. Because a *sequence* is simply another way of saying “a list of tensors,” we need to make sure we can convert our data accordingly. For example, a sentence is a sequence of words, but words are not tensors. The process of converting words to tensors, or more commonly vectors, is called *embedding*.

Last, we’ll put all these concepts together to implement the seq-to-seq model on real-world data. The data will come from thousands of conversations from movie scripts.

We can hit the ground running with the following code listing. Open a new Python file and start copying listing 11.1 to set up constants and placeholders. We will define the shape of the placeholder to be `[None, seq_size, input_dim]`, where `None` means the size is dynamic, because the batch-size may change, `seq_size` is the length of the sequence, and `input_dim` is the dimension of each sequence item.

Listing 11.1 Setting up constants and placeholders

```
import tensorflow as tf  ##A

input_dim = 1  ##B
seq_size = 6  ##C

input_placeholder = tf.placeholder(dtype=tf.float32,
                                  shape=[None, seq_size, input_dim])
```

#A all we need is TensorFlow
#B dimension of each sequence element
#C maximum length of sequence

To generate an RNN cell like the one in figure 11.2, TensorFlow provides a helpful `LSTMCell` class. Listing 11.2 shows how to use it and extract the outputs and states from the cell. Just for convenience, the listing defines a helper function called `make_cell` to set up the LSTM RNN cell. Remember, just defining a cell isn’t enough: you also need to call `tf.nn.dynamic_rnn` on it to set up the network.

Listing 11.2 Making a simple RNN cell

```
def make_cell(state_dim):
    return tf.contrib.rnn.LSTMCell(state_dim)  ##A

with tf.variable_scope("first_cell") as scope:
    cell = make_cell(state_dim=10)
    outputs, states = tf.nn.dynamic_rnn(cell,  ##B
                                       input_placeholder,  ##C
                                       dtype=tf.float32)
```

#A Check out `tf.contrib.rnn` documentation for other types of cells, such as GRU
#B There will be two generated results: outputs and states

#C This is the input sequence to the RNN

You might remember from previous chapters that we can improve a neural network's complexity by adding more and more hidden layers. More layers means more parameters, and that likely means the model can represent more functions; it's more flexible.

You know what? We can just keep stacking cells on top of each other. There's nothing stopping us, really. Doing so makes the model more complex, so perhaps this two-layered RNN model will perform better because it is more expressive. Figure 11.3 shows two cells stacked together.

WARNING The more flexible the model, the more likely it will overfit the training data.

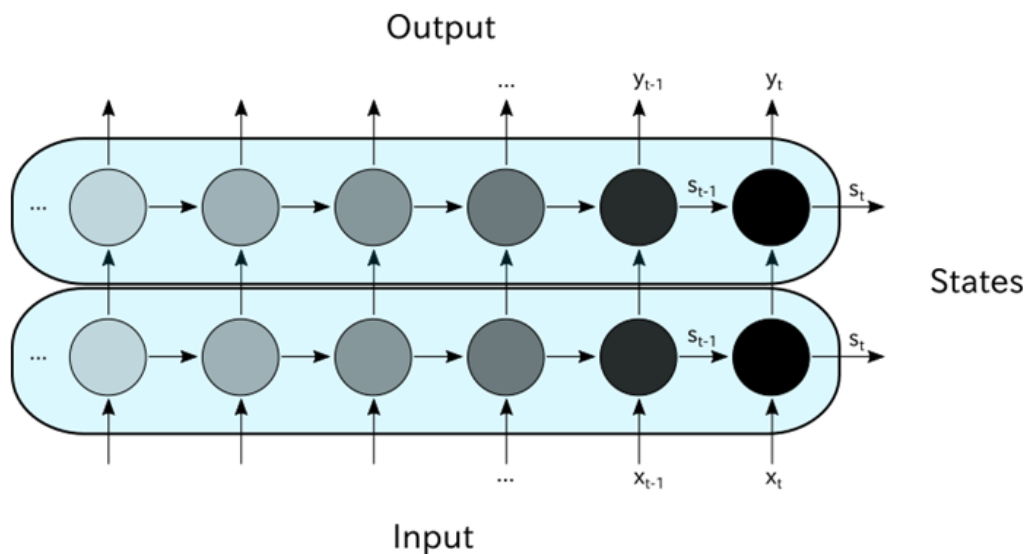


Figure 11.3 You can stack RNN cells together to form a more complicated architecture.

In TensorFlow, we can intuitively implement this two-layered RNN network. First, you create a new variable scope for the second cell. To stack RNNs together, you can pipe the output of the first cell to the input of the second cell. Listing 11.3 shows how to do exactly this.

Listing 11.3 Stacking two RNN cells

```
with tf.variable_scope("second_cell") as scope: //#A
    cell2 = make_cell(state_dim=10)
    outputs2, states2 = tf.nn.dynamic_rnn(cell2,
                                         outputs, //#B
                                         dtype=tf.float32)
```

#A defining a variable scope helps avoid run-time errors due to variable reuse

#B input to this cell will be the other cell's output

What if we wanted four layers of RNNs? Or 10? For example, figure 11.4 shows four RNN cells stacked together.

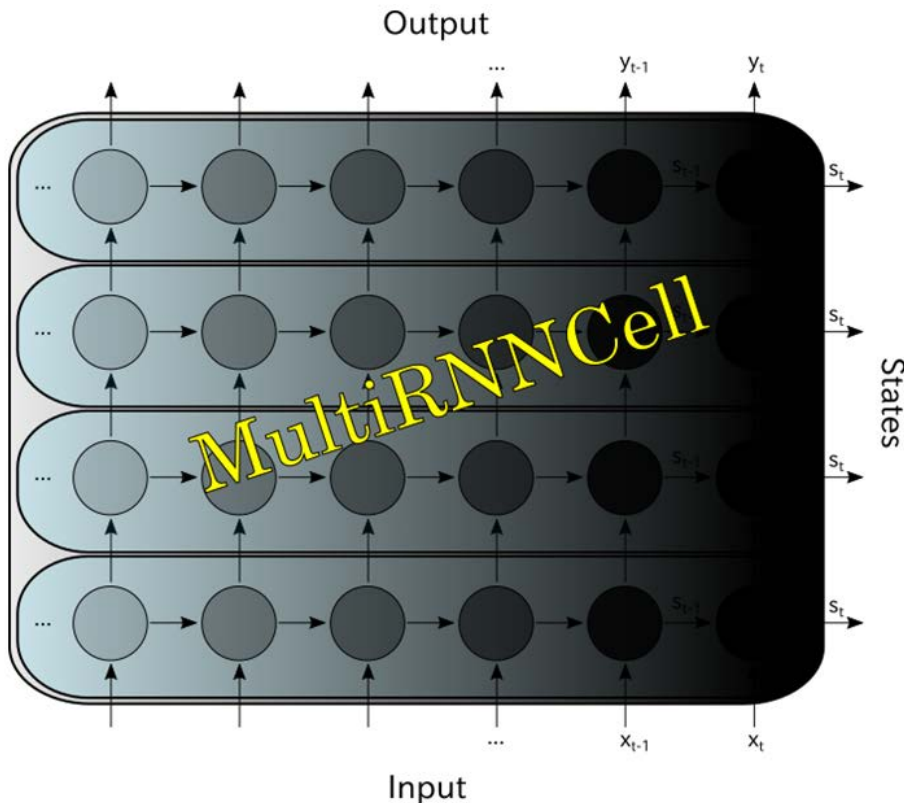


Figure 11.4 TensorFlow lets you stack as many RNN cells as you want.

There's a useful shortcut for stacking cells that the TensorFlow library supplies, called `MultiRNNCell`. Listing 11.4 shows how to use this helper function to build arbitrarily large RNN cells.

Listing 11.4 Using `MultiRNNCell` to stack multiple cells

```
def make_multi_cell(state_dim, num_layers):
    cells = [make_cell(state_dim) for _ in range(num_layers)] // #A
    return tf.contrib.rnn.MultiRNNCell(cells)

multi_cell = make_multi_cell(state_dim=10, num_layers=4)
outputs4, states4 = tf.nn.dynamic_rnn(multi_cell,
```

```
input_placeholder,
dtype=tf.float32)
```

#A the for-loop syntax is the preferred way to construct a list of RNN cells

So far, we've grown RNNs vertically by piping outputs of one cell to the inputs of another. In the seq-to-seq model, we'll want one RNN cell to process the input sentence, and another RNN cell to process the output sentence. In order to communicate between the two cells, we can also connect RNNs horizontally by connecting states from cell to cell, as shown in figure 11.5

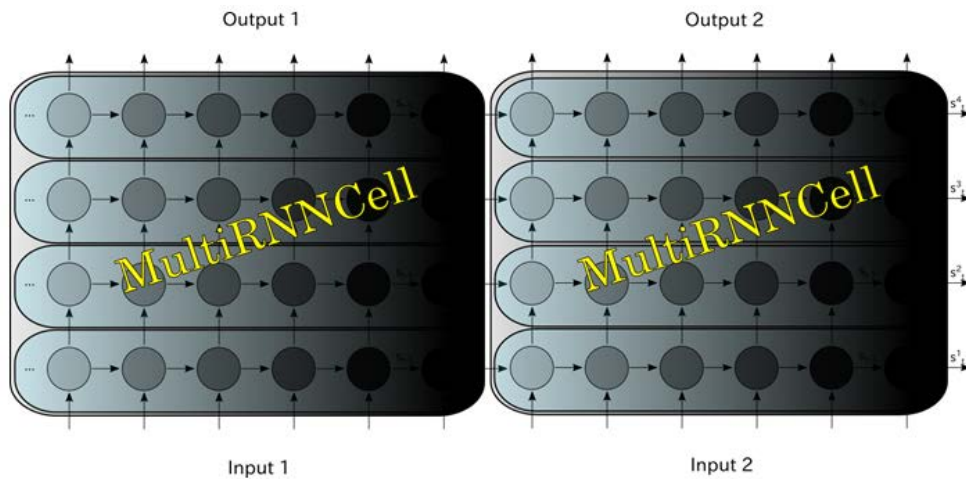


Figure 11.5 You can use the last states of the first cell as the next cell's initial state. This model can learn mapping from an input sequence to an output sequence. The model is called seq-to-seq.

So far, we've stacked RNN cells vertically and also connected them horizontally, vastly increasing the number of parameters in the network! Is this utter blasphemy? Yes. We've build a monolithic architecture by composing RNNs every which way. But there is some method to this madness, because this insane neural network architecture is the backbone of the *seq-to-seq model*.

As you can see in figure 11.5, It appears the seq-to-seq model has two input sequences and two output sequences. However, only Input 1 will be used for the input sentence and only Output 2 will be used for the output sentence.

You may be wondering what to do with the other two sequences. Strangely enough, the Output 1 sequence is entirely unused by the seq-to-seq model. And, as we'll see, the Input 2 sequence is crafted using some of Output 2 data, in a feedback loop.

Our training data for designing a chat-bot will be pairs of input and output sentences, so we will need to better understand how to embed words into a tensor. The next section covers how to do so in TensorFlow.

EXERCISE 11.2 Sentences may be represented by a sequence of characters or words, but can you think of other sequential representations of sentences?

11.3 Vector representation of symbols

Words or letters are symbols, and converting symbols to numeric values is easy in TensorFlow. For example, let's say we have four words in our vocabulary: *word₀*: "the", *word₁*: "fight", *word₂*: "wind", and *word₃*: "like".

Let's say we want to find the embeddings for the sentence, "Fight the wind." The symbol "fight" is located at index 1 of the lookup-table, "the" at index 0, and "wind" at index 2. So if we want to find the embedding of the word "fight", we have to refer to its index, which is 1, and consult the lookup table at index 1 to identify the embedding value.

In our first example, each word is associated with numbers, as shown in figure 11.6.

Word	Number
'the'	17
'fight'	22
'wind'	35
'like'	51

Figure 11.6 A mapping from symbols to scalars

Listing 11.5 shows how you can define such a mapping between symbols and numeric values using TensorFlow code.

Listing 11.5 Defining a lookup table of scalars

```
embeddings_0d = tf.constant([17, 22, 35, 51])
```

Or maybe, they're associated with vectors, as shown in figure 11.7 This is often the preferred method of representing words. You can find a thorough tutorial on vector representation of words on the official TensorFlow docs: <https://www.tensorflow.org/tutorials/word2vec>.

Word	Vector
'the '	[1, 0, 0, 0]
'fight'	[0, 1, 0, 0]
'wind'	[0, 0, 1, 0]
'like'	[0, 0, 0, 1]

Figure 11.7 A mapping from symbols to vectors

We can implement the mapping between words and vectors in TensorFlow, as shown in listing 11.6.

Listing 11.6 Defining a lookup table of 4D vectors

```
embeddings_4d = tf.constant([[1, 0, 0, 0],
                             [0, 1, 0, 0],
                             [0, 0, 1, 0],
                             [0, 0, 0, 1]])
```

This may sound over the top, but you can represent a symbol by a tensor of any rank you want, not just numbers (rank 0) or vectors (rank 1). In figure 11.8, we're mapping symbols to tensors of rank 2.

Word	Tensor
'the '	[[1, 0], [0, 0]]
'fight'	[[0, 1], [0, 0]]
'wind'	[[0, 0], [1, 0]]
'like'	[[0, 0], [0, 1]]

Figure 11.8 A mapping from symbols to tensors

Listing 11.7 shows how to implement this mapping of words to matrices in TensorFlow.

Listing 11.7 Defining a lookup table of tensors

```
embeddings_2x2d = tf.constant([[[1, 0], [0, 0]],
                                 [[0, 1], [0, 0]],
                                 [[0, 0], [1, 0]]])
```

```
[[0, 0], [0, 1]])
```

The `embedding_lookup` function provided by TensorFlow is an optimized way to access embedding by indices, as shown in listing 11.8.

Listing 11.8 Looking up the embeddings

```
ids = tf.constant([1, 0, 2])  ##A
lookup_0d = sess.run(tf.nn.embedding_lookup(embeddings_0d, ids))
print(lookup_0d)

lookup_4d = sess.run(tf.nn.embedding_lookup(embeddings_4d, ids))
print(lookup_4d)

lookup_2x2d = sess.run(tf.nn.embedding_lookup(embeddings_2x2d, ids))
print(lookup_2x2d)
```

#A Words corresponding to “fight”, “the”, and “wind”

In reality, the embedding matrix is not something you ever have to hard-code. These listings were for you to understand the ins and outs of the `embedding_lookup` function in TensorFlow, because we’ll be using it heavily soon. The actual embedding look-up table will be learned automatically over time by training the neural network. You start by defining a random, normally-distributed, look-up table. Then, TensorFlow’s optimizer will adjust the matrix values to minimize the cost.

EXERCISE 11.3 Follow the official TensorFlow word2vec tutorial to get more familiar with embeddings: <https://www.tensorflow.org/tutorials/word2vec>

11.4 Putting it all together

The first step in using natural language input into a neural network is to decide a mapping between symbols and integer indices. Two common ways to represent sentences is by a sequence of *letters* or a sequence of *words*. Let’s say, for simplicity, that we’re dealing with sequences of letters, so we’ll need to build a mapping between characters and integer indices.

BY THE WAY The official code repository is available on GitHub through the following url: https://github.com/BinRoot/TensorFlow-Book/blob/master/ch11_seq2seq/Concept03_seq2seq.ipynb. From there, you can get the code running without needing to copy and paste from the book.

Listing 11.9 shows how to build mappings between integers and characters. If you feed this function a list of strings, it will produce two dictionaries, representing the mappings.

Listing 11.9 Extract character vocab

```
def extract_character_vocab(data):
    special_symbols = ['<PAD>', '<UNK>', '<GO>', '<EOS>']
    set_symbols = set([character for line in data for character in line])
    all_symbols = special_symbols + list(set_symbols)
```



```

int_to_symbol = {word_i: word
                 for word_i, word in enumerate(all_symbols)}
symbol_to_int = {word: word_i
                 for word_i, word in int_to_symbol.items()}

return int_to_symbol, symbol_to_int

input_sentences = ['hello stranger', 'bye bye'] //#A
output_sentences = ['hiya', 'later alligator'] //#B

input_int_to_symbol, input_symbol_to_int =
    extract_character_vocab(input_sentences)

output_int_to_symbol, output_symbol_to_int =
    extract_character_vocab(output_sentences)

```

#A list of input sentences for training

#B list of corresponding output sentences for training

Next up, we'll define all our hyper-parameters and constants in listing 11.10. These are typically values you can tune by hand through trial and error. Typically, greater values for the number of dimensions or layers results in a more complex model, which is rewarding if you have big data, fast processing power, and lots of time.

Listing 11.10 Hyper-parameters

```

NUM_EPOCHS = 300 //#A
RNN_STATE_DIM = 512 //#B
RNN_NUM_LAYERS = 2 //#C
ENCODER_EMBEDDING_DIM = DECODER_EMBEDDING_DIM = 64 //#D

BATCH_SIZE = int(32)
LEARNING_RATE = 0.0003

INPUT_NUM_VOCAB = len(input_symbol_to_int) //#E
OUTPUT_NUM_VOCAB = len(output_symbol_to_int) //#E

```

#A number of epochs

#B RNNs' hidden dimension size

#C RNNs' number of stacked cells

#D embedding dimension of sequence elements for encoder and decoder

#E batch size

#F it's possible to have different vocabularies between the encoder and decoder

Let's list all placeholders next. As you can see in listing 11.11, the placeholders nicely organize the input and output sequences necessary to train the network. We'll have to track both the sequences and their lengths. For the decoder part, we'll also need to compute the maximum sequence length. The `None` value in the shape of these placeholders means the tensor may take on an arbitrary size in that dimension. For example, the batch-size may vary in each run. However, just for simplicity, we'll keep the batch-size the same at all times.

Listing 11.11 Placeholders

```

# Encoder placeholders
encoder_input_seq = tf.placeholder( //#A
    tf.int32,
    [None, None], //#B
    name='encoder_input_seq'
)

encoder_seq_len = tf.placeholder( //#C
    tf.int32,
    (None,), //#D
    name='encoder_seq_len'
)

# Decoder placeholders
decoder_output_seq = tf.placeholder( //#E
    tf.int32,
    [None, None], //#F
    name='decoder_output_seq'
)

decoder_seq_len = tf.placeholder( //#G
    tf.int32,
    (None,), //#H
    name='decoder_seq_len'
)

max_decoder_seq_len = tf.reduce_max( //#I
    decoder_seq_len,
    name='max_decoder_seq_len'
)

```

#A sequence of integers for the encoder's input
#B shape is batch-size x sequence-length
#C lengths of sequences in a batch
#D shape is dynamic because length of a sequence can change
#E sequence of integers for the decoder's output
#F shape is batch-size x sequence-length
#G lengths of sequences in a batch
#H shape is dynamic because length of a sequence can change
#I maximum length of a decoder sequence in a batch

Let's define some helper functions to construct RNN cells. These functions, shown in listing 11.12, should appear familiar to you from the previous section.

Listing 11.12 Helper function to build RNN cells

```

def make_cell(state_dim):
    lstm_initializer = tf.random_uniform_initializer(-0.1, 0.1)
    return tf.contrib.rnn.LSTMCell(state_dim, initializer=lstm_initializer)

def make_multi_cell(state_dim, num_layers):
    cells = [make_cell(state_dim) for _ in range(num_layers)]
    return tf.contrib.rnn.MultiRNNCell(cells)

```

We'll build the encoder and decoder RNN cells using the helper functions we've just defined. As a reminder, I've copied the seq-to-seq model for you in figure 11.9, to visualize the encoder and decoder RNNs.

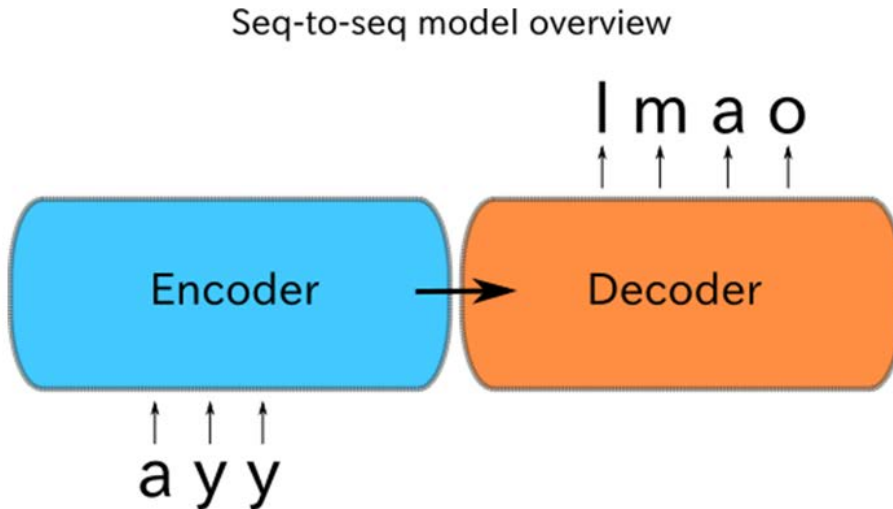


Figure 11.9 The seq-to-seq model learns a transformation between an input sequence to an output sequence using an encoder RNN and a decoder RNN.

Let's talk about the encoder cell part first, because in listing 11.13 we will build the encoder cell. The produced states of the encoder RNN will be stored in a variable called `encoder_state`. RNNs also produce an output sequence, but we don't really need access to that in a standard seq-to-seq model, so we can ignore it or delete it.

It's also typical to convert letters or words in a vector representation, often called *embedding*. TensorFlow provides a handy function called `embed_sequence` that can help us embed the integer representation of symbols. Figure 11.10 shows the how the input the encoder RNN accepts numeric values from an embedding look-up table. We can see it in action in the beginning of listing 11.13.

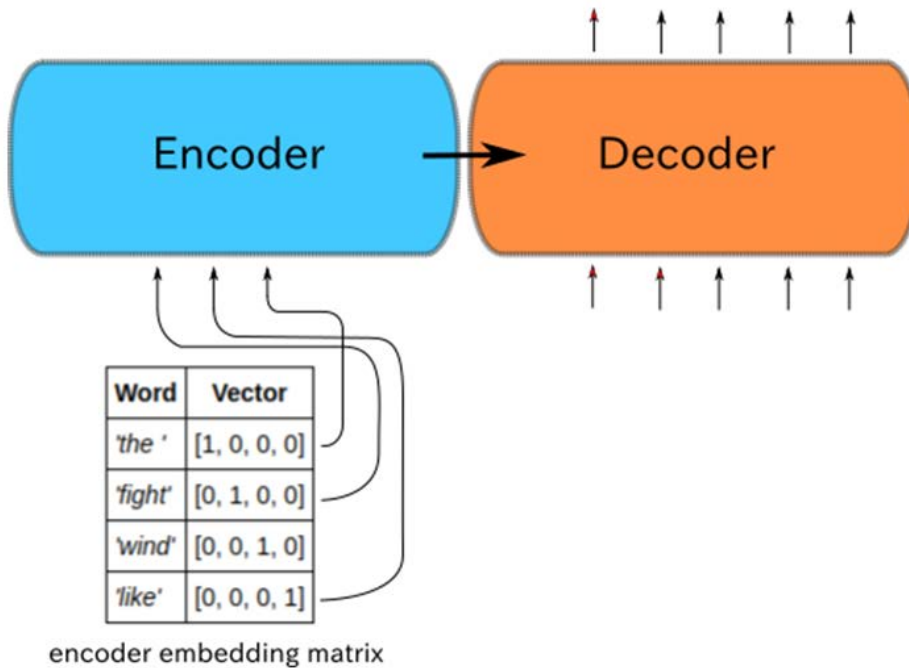


Figure 11.10 The RNNs only accept sequences of numeric values as input or output, so we will convert our symbols to vectors. In this case, the symbols are words, such as “the”, “fight”, “wind”, or “like”. Their corresponding vectors are associated in the embedding matrix.

Listing 11.13 Encoder embedding and cell

```
# Encoder embedding

encoder_input_embedded = tf.contrib.layers.embed_sequence(
    encoder_input_seq,      //#A
    INPUT_NUM_VOCAB,      //#B
    ENCODER_EMBEDDING_DIM //#C
)

# Encoding output

encoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)

encoder_output, encoder_state = tf.nn.dynamic_rnn(
    encoder_multi_cell,
    encoder_input_embedded,
    sequence_length=encoder_seq_len,
    dtype=tf.float32
)

del(encoder_output) //#D
```

```
#A input seq of numbers (row indices)
#B rows of embedding matrix
#C cols of embedding matrix
#D we don't need to hold on to that value
```

The decoder RNN's output is a sequence of numeric values representing a natural language sentence and a special symbol to represent that the sequence has ended. We'll label this end-of-sequence symbol as <EOS>. Figure 11.11 visualizes this process for you.

The input sequence to the decoder RNN will look very similar to the decoder's output sequence, except instead of having the <EOS> (end of sequence) special symbol in the end of each sentence, it will instead have a <GO> special symbol in the front. That way, once the decoder reads its input left-to-right, it starts out with no extra information about the answer, making it a robust model.

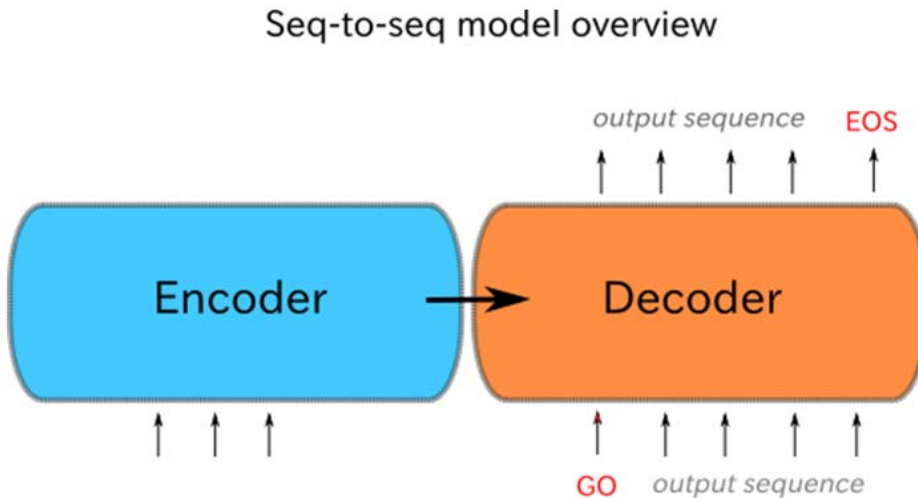


Figure 11.11 The decoder's input is prefixed with a special "<GO>" symbol, whereas the output is suffixed by a special "<EOS>" symbol.

Listing 11.14 shows how to correctly perform these slicing and concatenating operations. The newly constructed sequence for the decoder's input will be called `decoder_input_seq`. We will use TensorFlow's `tf.concat` operation to glue together matrices. In the listing, we'll end up defining a `go_prefixes` matrix, which will be a column vector containing only the <GO> symbol.

Listing 11.14 Preparing input sequences to the decoder

```
decoder_raw_seq = decoder_output_seq[:, :-1] // #A
go_prefixes = tf.fill([BATCH_SIZE, 1], output_symbol_to_int['<GO>']) // #B
decoder_input_seq = tf.concat([go_prefixes, decoder_raw_seq], 1) // #C
```

```
#A crop the matrix by ignoring the very last column
#B create a column vector of '<GO>' symbols
#C concatenate the '<GO>' vector to the beginning of the cropped matrix
```

Now let's construct the decoder cell. As shown in listing 11.15, we'll first embed the decoder sequence of integers into a sequence of vectors, called `decoder_input_embedded`.

The embedded version of the input sequence will be fed to the decoder's RNN, so go ahead and create the decoder RNN cell. One more thing, we'll need a layer to map the output of the decoder to a one-hot representation of the vocabulary, which we call the `output_layer`. The process in setting up the decoder starts out to be very similar to that with the encoder.

Listing 11.15 Decoder embedding and cell

```
decoder_embedding = tf.Variable(tf.random_uniform([OUTPUT_NUM_VOCAB,
                                                DECODER_EMBEDDING_DIM]))
decoder_input_embedded = tf.nn.embedding_lookup(decoder_embedding,
                                                decoder_input_seq)

decoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)

output_layer_kernel_initializer =
    tf.truncated_normal_initializer(mean=0.0, stddev=0.1)
output_layer = Dense(
    OUTPUT_NUM_VOCAB,
    kernel_initializer = output_layer_kernel_initializer
)
```

Okay, here's where things get weird. There will be two ways to retrieve the decoder's output: during training and during inference. The training decoder will only be used during training, whereas the inference decoder will be used for testing on never-before-seen data.

The reason for having two ways to obtain an output sequence is due to the fact that during training, we have the ground-truth data available to us, so we can use information about the known output to help speed up the learning process. But during inference, we have no ground-truth output labels, so we must resort to making inferences only using the input sequence.

Follow listing 11.16 to implementing the training decoder. We'll feed `decoder_input_seq` into the decoder's input, using something called a `TrainingHelper`. This helper op essentially manager the input to the decoder RNN for us.

Listing 11.16 Decoder output (Training)

```
with tf.variable_scope("decode"):

    training_helper = tf.contrib.seq2seq.TrainingHelper(
        inputs=decoder_input_embedded,
        sequence_length=decoder_seq_len,
        time_major=False
    )

    training_decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_multi_cell,
```

```

        training_helper,
        encoder_state,
        output_layer
    )

    training_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(
        training_decoder,
        impute_finished=True,
        maximum_iterations=max_decoder_seq_len
    )

```

If we care to obtain an output from the seq-to-seq model on test data, we no longer have access to `decoder_input_seq`. Why? Well, the decoder input sequence is derived from the decoder output sequence, which is only available with the training dataset.

Follow along to listing 11.17 to implement the decoder output op for the inference case. Here again, we will use a helper op to feed the decoder an input sequence.

Listing 11.17 Decoder output (Inference)

```

with tf.variable_scope("decode", reuse=True):
    start_tokens = tf.tile(
        tf.constant([output_symbol_to_int['<GO>']],
                    dtype=tf.int32),
        [BATCH_SIZE],
        name='start_tokens')

    //#A
    inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
        embedding=decoder_embedding,
        start_tokens=start_tokens,
        end_token=output_symbol_to_int['<EOS>']
    )

    //#B
    inference_decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_multi_cell,
        inference_helper,
        encoder_state,
        output_layer
    )

    //#C
    inference_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(
        inference_decoder,
        impute_finished=True,
        maximum_iterations=max_decoder_seq_len
    )

```

#A Helper for the inference process.

#B Basic decoder

#C Perform dynamic decoding using the decoder

Compute the cost using TensorFlow's `sequence_loss` method. We'll need access to the inferred decoder output sequence and the ground-truth output sequence. See listing 11.18 for how to define the cost function in code.

Listing 11.18 Cost function

```

##A
training_logits =
    tf.identity(training_decoder_output_seq.rnn_output, name='logits')
inference_logits =
    tf.identity(inference_decoder_output_seq.sample_id, name='predictions')

##B
masks = tf.sequence_mask(
    decoder_seq_len,
    max_decoder_seq_len,
    dtype=tf.float32,
    name='masks'
)

##C
cost = tf.contrib.seq2seq.sequence_loss(
    training_logits,
    decoder_output_seq,
    masks
)

```

#A Rename the tensors for our convenience
#B Create the weights for `sequence_loss`
#C Use TensorFlow's built-in sequence loss function

Last, let's call an optimizer to minimize the cost. But, we'll do one trick you might have never seen before. In very deep networks like this one, we need to limit extreme gradient change to ensure the gradient does not change too dramatically, a technique called *gradient clipping*. Listing 11.19 shows you how to do so.

EXERCISE 11.4 Try the seq-to-seq model without gradient clipping to experience the difference.

Listing 11.19 Optimizer

```

optimizer = tf.train.AdamOptimizer(LEARNING_RATE)

gradients = optimizer.compute_gradients(cost)
capped_gradients = [(tf.clip_by_value(grad, -5., 5.), var) ##A
                    for grad, var in gradients if grad is not None]
train_op = optimizer.apply_gradients(capped_gradients)

```

#A gradient clipping

That concludes the seq-to-seq model implementation. In general, the model is ready to be trained once you have set up the optimizer, as in the previous listing. We can create a session and run the `train_op` with batches of training data to learn the parameters of the model.

Oh right, we need training data from someplace! How can we obtain thousands of pairs of input and output sentences? Fear not, the next section covers exactly that.

11.5 Gathering dialogue data

The Cornell Movie Dialogue corpus (https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html) is a dataset of over 220 thousand conversations from over 600 movies. You can download the zip file from the official webpage.

WARNING Because there's a huge amount of data, you can expect the training algorithm to take a very long time. If your TensorFlow library is configured to only use the CPU, it might take an entire day to train. On a GPU, training this network may take 30 minutes to an hour.

An example of a small snippet of back-and-forth conversation between two people (A and B) is the following:

A: They do not!

B: They do too!

A: Fine.

Because the goal of the chatbot is to produce intelligent output for every possible input utterance, we'll structure our training data based on contingent pairs of conversation. So in our example, the dialogue generates the following pairs of input and output sentences.

- "They do not!" → "They do too!"
- "They do too!" → "Fine."

For your convenience, I've already processed the data and made it available for you online. You can find it through the following URL https://github.com/BinRoot/TensorFlow-Book/tree/master/ch11_seq2seq. Once downloaded, you can run the following code in listing 11.20, which uses the `load_sentences` helper function from the GitHub repo under the `Concept03_seq2seq.ipynb` Jupyter notebook.

Listing 11.20 Training the model

```
input_sentences = load_sentences('data/words_input.txt') // #A
output_sentences = load_sentences('data/words_output.txt') // #B

input_seq = [
    [input_symbol_to_int.get(symbol, input_symbol_to_int['<UNK>'])
     for symbol in line] // #C
    for line in input_sentences // #D
]

output_seq = [
    [output_symbol_to_int.get(symbol, output_symbol_to_int['<UNK>'])
     for symbol in line] + [output_symbol_to_int['<EOS>']] // #E
```

```

    for line in output_sentences //#F
]

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver() //#G

for epoch in range(NUM_EPOCHS + 1): //#H

    for batch_idx in range(len(input_sentences) // BATCH_SIZE): //#I

        input_data, output_data = get_batches(input_sentences, //#J
                                              output_sentences,
                                              batch_idx)

        input_batch, input_lengths = input_data[batch_idx]
        output_batch, output_lengths = output_data[batch_idx]

        _, cost_val = sess.run( //#K
            [train_op, cost],
            feed_dict={
                encoder_input_seq: input_batch,
                encoder_seq_len: input_lengths,
                decoder_output_seq: output_batch,
                decoder_seq_len: output_lengths
            }
        )

saver.save(sess, 'model.ckpt')
sess.close()

```

#A load the input sentences as a list of string
#B load the corresponding output sentences the same way
#C Loop through the letters
#D Loop through the lines of text
#E Append the “EOS” symbol to the end of the output data
#F Loop through the lines
#G It’s a good idea to save the learned parameters
#H Loop through the epochs
#I Loop by the number of batches
#J Get input and output pairs for the current batch
#K Run the optimizer on the current batch

Because you saved the model parameters to file, you can easily load it onto another program and query the network for responses to new input. Run the `inference_logits` op to obtain chatbot response.

You can find the whole code, up to date, on the book’s official repository: https://github.com/BinRoot/TensorFlow-Book/tree/master/ch11_seq2seq.

11.6 Summary

In this chapter, you learned the following:

- In this chapter we built a seq-to-seq neural network by putting to work all the

TensorFlow knowledge we've acquired from the book so far.

- We learned how to embed natural language in TensorFlow.
- We used RNNs as a building block for a more interesting model
- After training the model on examples of dialogue from movie scripts, we were able to treat the algorithm like a chatbot, inferring natural language responses from natural language input.

12

Utility landscape

This chapter covers:

- **Neural network for ranking**
- **Image embedding using VGG-16**
- **Visualizing utility**

A house-hold vacuuming robot, like the Roomba, needs sensors to “see” the world. The ability to process sensory input enables robots to adjust their model of the world around them. In the case of the vacuum-cleaner robot, the furniture in the room may change day to day, so the robot must be able to adapt to chaotic environments.

Let’s say you own a futuristic house-maid robot, which comes with a few basic skills but also with the ability to learn new skills from human demonstrations. For example, maybe you would like to teach it how to fold clothes.

Teaching a robot how to accomplish a new task is a tricky problem. Some immediate questions come to mind:

- Should the robot simply mimic a human’s sequence of actions? Such a process is referred to as *imitation learning*.
- How do robot’s arms and joints match up to human poses? This dilemma is often referred to as the *correspondence problem*.

POP QUIZ The goal of imitation learning is for the robot to reproduce the action-sequences of the demonstrator. This sounds good on paper, but what are the limitations of such an approach?

In this chapter, we’re going to model a task from human demonstrations while avoiding both imitation learning and the correspondence problem. Lucky for you! We’ll achieve this by studying a way to rank states of the world using something called a *utility function*, which is a

function that takes a state and returns a real value that represents its desirability. Not only will we steer away from imitation as a measure of success, but we will also bypass the complications of mapping a robot's set of actions to that of a human's (otherwise known as the correspondence problem).

In the following section, you will learn how to implement a utility function over the states of the world obtained through videos of human demonstrations of a task. The learned utility function is a model of preferences.

We explore the task of teaching a robot how to fold articles of clothing. A wrinkled article of clothing is almost certainly in a configuration that has never before been seen. As shown in figure 12.1, the utility framework has no limitations on the size of the state-space. The preference model is trained specifically on videos of people folding t-shirts in various ways.

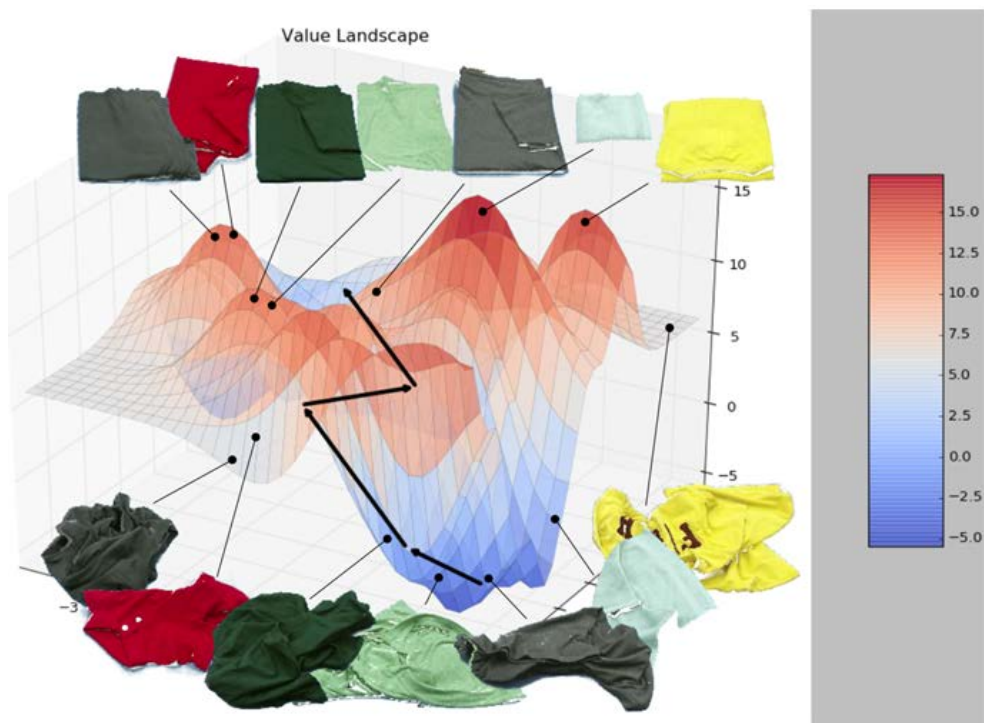


Figure 12.1 Wrinkled clothes are less favorable than well-folded clothes. This diagram shows how we might score each state of a cloth, where higher scores represent a more favorable state.

The utility function generalizes across states (wrinkled t-shirt in novel configuration versus folded t-shirt in familiar configuration) and reuses knowledge across clothes (t-shirt folding versus pants folding).

We further motivate the practical applications of a good utility function with the following argument: in real-world situations, not all visual observations are optimized toward learning a task. A teacher demonstrating a skill may perform irrelevant, incomplete, or even incorrect actions, yet humans are capable of ignoring the mistakes.

When a robot watches human demonstrations, we want it to understand the causal relationships that go into achieving a task. Our work enables the learning phase to be interactive, where the robot is actively skeptic of human behavior, to refine the training data.

To accomplish this, we first learn a utility function from a small number of videos to rank the preferences of various states. Then, when the learner is shown a new instance of a skill through human demonstration, it consults the utility function to verify that the expected utility increases over time. Lastly, the learner interrupts the human demonstration to ask if the action was essential for learning the skill.

12.1 Preference model

We assume human preferences are derived from a *utilitarian* perspective, meaning a number simply determines the rank of items. For example, suppose we surveyed people to rank the fanciness of various foods (such as steak, hotdog, shrimp cocktail, and burger).

Figure 12.2 shows a couple possible rankings between pairs of food. As you might expect, steak is ranked higher than hotdog, and shrimp cocktail higher than burger in the fanciness scale.

Ranking food by fanciness



Figure 12.2 This is a possible set of pair-wise rankings between objects. Specifically, there are four food items, and we would like to rank them by fanciness, so we employ two pair-wise ranking decisions: steak is a more fancy meal than hotdogs and shrimp cocktail is a more fancy meal than burgers.

Fortunately for the individual being surveyed, not every pair of items needs to be ranked. For example, it might not be so obvious which is fancier between hotdog and burger, or between steak and shrimp cocktail. There's a lot of room for disagreement there.

If a state s_1 has a higher utility than another state s_2 , then the corresponding ranking is denoted $s_1 > s_2$, implying the utility of s_1 is greater than the utility of s_2 .

Each video demonstration contains a sequence of n states s_0, s_1, \dots, s_n , which offers $n(n-1)/2$ possible ordered pairs ranking constraints. Let's implement our own neural network capable of ranking. Open a new source file, and let's start by following listing 12.1 to import the relevant libraries. We're about to create a neural network to learn a utility function based on pairs of preferences.

Listing 12.1 Import relevant libraries

```
import tensorflow as tf
import numpy as np
import random

%matplotlib inline
import matplotlib.pyplot as plt
```

To learn a neural network for ranking states based on a utility score, we will need some training data. Let's just create some dummy data to begin with. We'll replace it with something more realistic later. Let's reproduce the two-dimensional data in figure 12.3 by following listing 12.2.

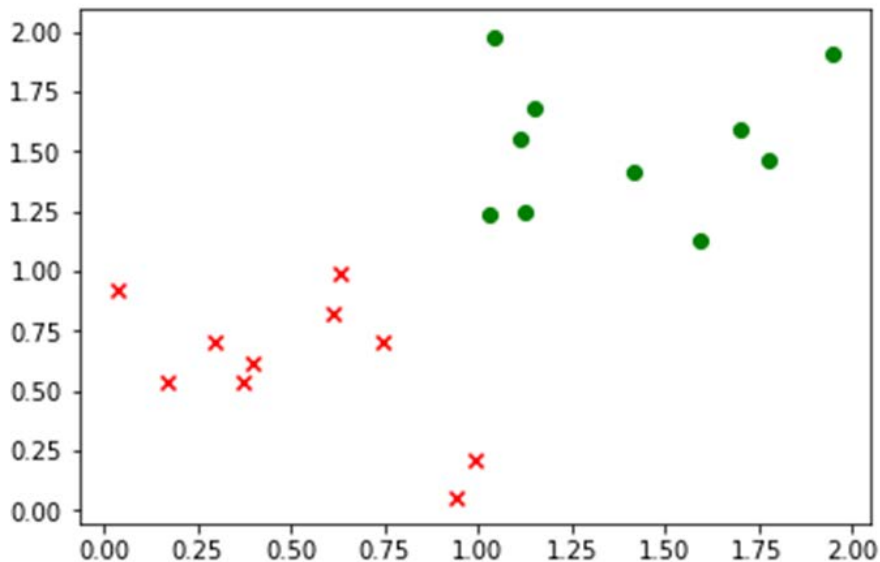


Figure 12.3 Example data that we will work with. The green circles represent more favorable states whereas the red crosses represent less favorable states. There is an equal number of circles and crosses because the data comes in pairs, where each pair is a ranking like figure 12.2.

Listing 12.2 Generate dummy training data

```

n_features = 2  //#A

def get_data():
    data_a = np.random.rand(10, n_features) + 1  //#B
    data_b = np.random.rand(10, n_features)  //#C

    plt.scatter(data_a[:, 0], data_a[:, 1], c='r', marker='x')
    plt.scatter(data_b[:, 0], data_b[:, 1], c='g', marker='o')
    plt.show()

    return data_a, data_b

data_a, data_b = get_data()

```

#A We'll generate two-dimensional data so that we can easily visualize them

#B The set of points that should yield higher utility

#C The set of points that are less preferred

Next, we need to define hyperparameters. In this model, let's keep it simple by keeping the architecture shallow. We'll create a network with just one hidden layer, and the corresponding hyper-parameter that dictates the hidden layer's number of neurons is the following:

```
n_hidden = 10
```

The ranking neural network will receive pairwise input, so we'll need to have two separate placeholders, one for each part of the pair. Moreover, we'll create a placeholder to hold the dropout parameter value. Continue adding the following code in listing 12.3 to your script.

Listing 12.3 Placeholders

```

with tf.name_scope("input"):
    x1 = tf.placeholder(tf.float32, [None, n_features], name="x1")  //#A
    x2 = tf.placeholder(tf.float32, [None, n_features], name="x2")  //#B
    dropout_keep_prob = tf.placeholder(tf.float32, name='dropout_prob')

```

#A Input placeholder for preferred points

#B Input placeholder for not preferred points

The ranking neural network will contain only one hidden layer. In listing 12.4, we'll define the weights and biases, and then re-use these weights and biases on each of the two input placeholders.

Listing 12.4 Hidden layer

```

with tf.name_scope("hidden_layer"):
    with tf.name_scope("weights"):
        w1 = tf.Variable(tf.random_normal([n_features, n_hidden]), name="w1")
        tf.summary.histogram("w1", w1)
        b1 = tf.Variable(tf.random_normal([n_hidden]), name="b1")
        tf.summary.histogram("b1", b1)

    with tf.name_scope("output"):

```



```

h1 = tf.nn.dropout(tf.nn.relu(tf.matmul(x1,w1) + b1), keep_prob=dropout_keep_prob)
tf.summary.histogram("h1", h1)
h2 = tf.nn.dropout(tf.nn.relu(tf.matmul(x2, w1) + b1), keep_prob=dropout_keep_prob)
tf.summary.histogram("h2", h2)

```

The goal of the neural network is to calculate a score for the two inputs provided. In listing 12.5, we define the weights, biases, and fully connected architecture of the output layer of the network. We'll be left with two outputs vectors, `s1` and `s2`, representing the scores for the pairwise input.

Listing 12.5 Output layer

```

with tf.name_scope("output_layer"):
    with tf.name_scope("weights"):
        w2 = tf.Variable(tf.random_normal([n_hidden, 1]), name="w2")
        tf.summary.histogram("w2", w2)
        b2 = tf.Variable(tf.random_normal([1]), name="b2")
        tf.summary.histogram("b2", b2)

    with tf.name_scope("output"):
        s1 = tf.matmul(h1, w2) + b2  ##A
        s2 = tf.matmul(h2, w2) + b2  ##B

```

#A Utility score of input x1

#B Utility score of input x2

We'll assume that when training the neural network, `x1` should contain the less favorable items. That means `s1` should be scored lower than `s2`, meaning the difference between `s1` and `s2` should be negative. As listing 12.6 shows, the loss function tries to guarantee a negative difference using the softmax cross-entropy loss. We'll define a `train_op` to minimize the loss function.

Listing 12.6 Loss and optimizer

```

with tf.name_scope("loss"):
    s12 = s1 - s2
    s12_flat = tf.reshape(s12, [-1])

    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=tf.zeros_like(s12_flat),
        logits=s12_flat + 1)

    loss = tf.reduce_mean(cross_entropy)
    tf.summary.scalar("loss", loss)

with tf.name_scope("train_op"):
    train_op = tf.train.AdamOptimizer(0.001).minimize(loss)

```

Now, follow listing 12.7 to set up a TensorFlow session. This involves initializing all variables and also preparing TensorBoard debugging using a summary writer.

BY THE WAY We've used a summary writer before in the end of chapter 2, when we were first introduced to TensorBoard.

Listing 12.7 Prepare a session

```
sess = tf.InteractiveSession()
summary_op = tf.summary.merge_all()
writer = tf.summary.FileWriter("tb_files", sess.graph)
init = tf.global_variables_initializer()
sess.run(init)
```

We're ready to train the network! Run the `train_op` on the dummy data we generated to learn the parameters of the model.

Listing 12.8 Train the network

```
for epoch in range(0, 10000):
    loss_val, _ = sess.run([loss, train_op], feed_dict={x1:data_a, x2:data_b,
                                                         dropout_keep_prob:0.5}) //A
    if epoch % 100 == 0 :
        summary_result = sess.run(summary_op,
                                   feed_dict={x1:data_a, //B
                                               x2:data_b, //C
                                               dropout_keep_prob:1}) //D
        writer.add_summary(summary_result, epoch)
```

#A Training dropout keep_prob is 0.5
 #B Preferred points
 #C Not preferred points
 #D Testing dropout keep_prob should always be 1

Finally, let's visualize the learned score function. As shown in listing 12.9, append two-dimensional points to a list.

Listing 12.9 Preparing test data

```
grid_size = 10
data_test = []
for y in np.linspace(0., 1., num=grid_size): //A
    for x in np.linspace(0., 1., num=grid_size): //B
        data_test.append([x, y])
```

#A Loop through the rows
 #B Loop through the columns

We will run the `s1` op on the test data to obtain utility values of each state, and visualize it as seen in figure 12.4.

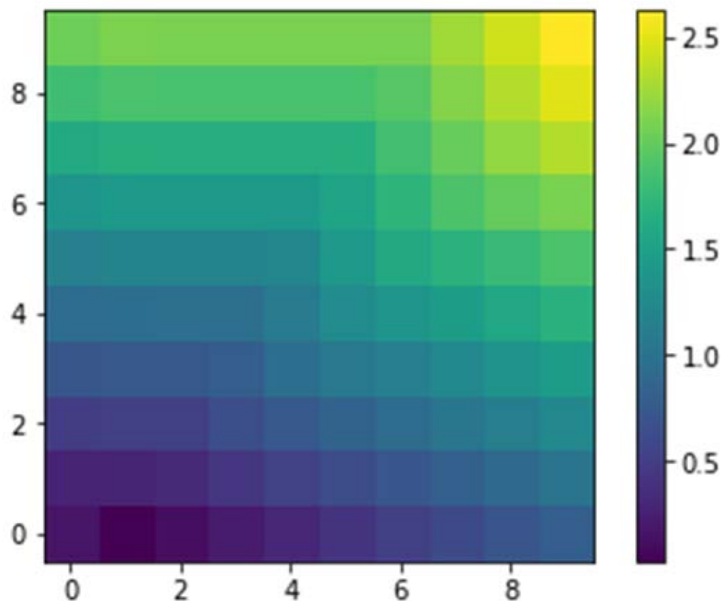


Figure 12.4 The landscape of scores learned by the ranking neural network

Follow along to listing 12.10 to generate the visualization shown in figure 12.4.

Listing 12.10 Visualize results

```
def visualize_results(data_test):
    plt.figure()
    scores_test = sess.run(s1, feed_dict={x1:data_test, dropout_keep_prob:1}) //#A
    scores_img = np.reshape(scores_test, [grid_size, grid_size]) //#B
    plt.imshow(scores_img, origin='lower')
    plt.colorbar()

visualize_results(data_test)
```

#A Compute the utility of all the points

#B Reshape the utilities to a matrix so we can visualize an image using matplotlib

12.2 Image embedding

In chapter 11, we summoned the hubris to feed a neural network natural-language sentences. We did so by converting words or letters in a sentence into numeric forms, such as vectors. For example, each symbol (whether it be a word or letter) was embedded into a vector using a lookup table.

POP QUIZ Why is a lookup table that converts a symbol into a vector representation called an embedding matrix?

Fortunately, images are already in a numeric form. They're represented as a matrix of pixels. If the image is grayscale, then perhaps the pixels take on scalar values indicating luminosity. For colored images, each pixel represents color intensities (usually three: red, green, and blue). Either way, an image can easily be represented by numeric data-structures, such as a Tensor, in TensorFlow.

EXERCISE 12.X Take a photo of a household object, such as a chair. Scale the image smaller and smaller until you can no longer identify the object. By what factor did you end up shrinking the image? What's the ratio of the number of pixels in the original image to the number of images in the smaller image? This ratio is a rough measure of redundancy in the data.

Feeding a neural network a large image, say of size 1280 x 720 (almost 1 million pixels), increases the number of parameters, and consequently surges the risk of overfitting the model. The pixels in an image are highly redundant, so we can try to somehow capture the essence of an image in a more succinct representation. Figure 12.5 shows the clusters formed in a two-dimensional embedding of images of clothes being folded.

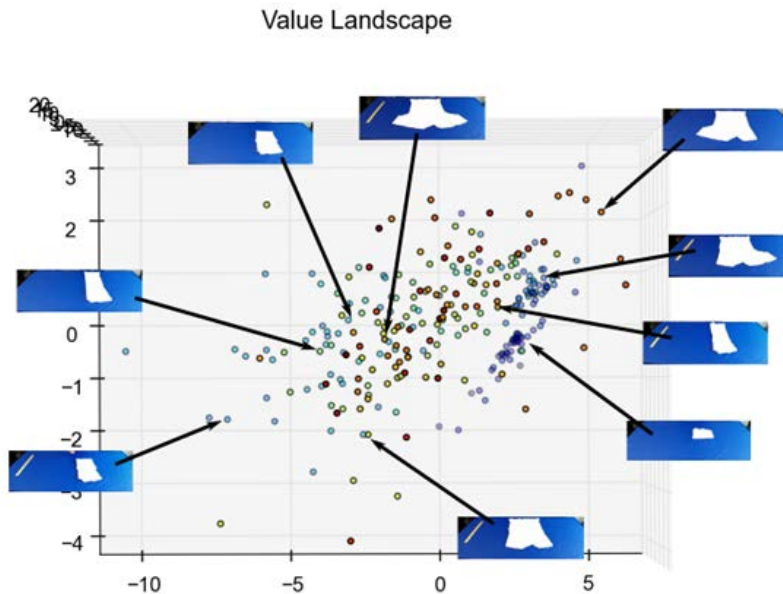


Figure 12.5 Images can be embedded into much lower dimensions, like 2-D as shown here. Notice how points representing similar states of a shirt occur in nearby clusters. Embedding images allows us to use the ranking neural network to learn a preference between the states of a cloth.

We saw in chapter 7 how to use autoencoders to lower the dimensionality of images. Another common way to accomplish low-dimensional embedding of images is by using the

penultimate layer of a deep convolutional neural network image classifier. Let's explore the latter in more detail.

Because designing, implementing, and learning a deep image classifier is not the primary focus of this chapter (see chapter 9 for CNNs), we will instead use an off-the-shelf pre-trained model. A common go-to image classifier that many computer vision research papers end up citing is called VGG-16.

In fact, there are many online implementations of VGG-16 for TensorFlow. I recommend using the one by Davi Frossard (<https://www.cs.toronto.edu/~frossard/post/vgg16/>). You can download the `vgg16.py` TensorFlow code and the `vgg16_weights.npz` pre-trained model parameters from his website, or alternatively from the book's official GitHub repo (<https://github.com/BinRoot/TensorFlow-Book>).

Figure 12.6 is a visualization of the VGG-16 neural network from Frossard's page. As you see, it's a very deep neural network, with many convolutional layers. The last few are the usual fully connected layers, and finally the output layer is a 1000-dimensional vector indicating the multi-class classification probabilities.

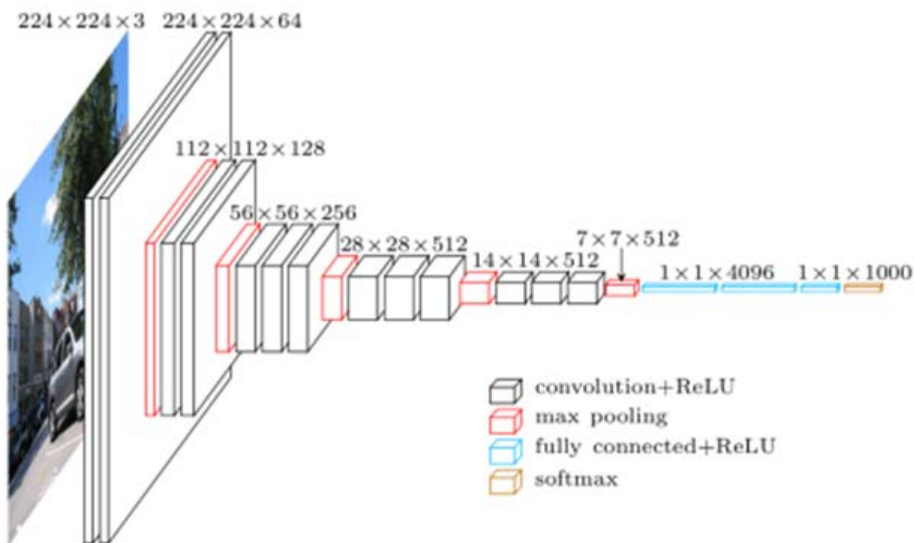


Figure 12.6 The VGG-16 architecture is a deep convolutional neural network used for classifying images. This particular diagram is by <https://www.cs.toronto.edu/~frossard/post/vgg16/>.

Learning how to navigate other people's code is an indispensable skill. First, make sure you have `vgg16.py` and `vgg16_weights.npz` downloaded and test that you're able to run the code using `python vgg16.py any_image.png`.

BY THE WAY You might need to install `scipy` and `Pillow` to get the `vgg16` demo code to run without issues. You can download both via `pip`.

Let's start by adding TensorBoard integration to visualize what's really going on in this code. In the main function, after creating a session variable `sess`, insert the following line of code.

```
my_writer = tf.summary.FileWriter('tb_files', sess.graph)
```

Now, running the classifier once again (`python vgg16.py my_image.png`) will generate a directory called `tb_files`, to be used by TensorBoard. You can run TensorBoard to visualize the computation graph of the neural network. The following command will run TensorBoard:

```
$ tensorboard --logdir=tb_files
```

Open TensorBoard in your browser, and navigate to the graphs tab to see the computation graph, as shown in figure 12.7. Notice how with a quick glance you can immediately get an idea of the types of layers involved in the network. Namely, the last three layers are fully connected dense layers, labelled by `fc1`, `fc2`, and `fc3`.

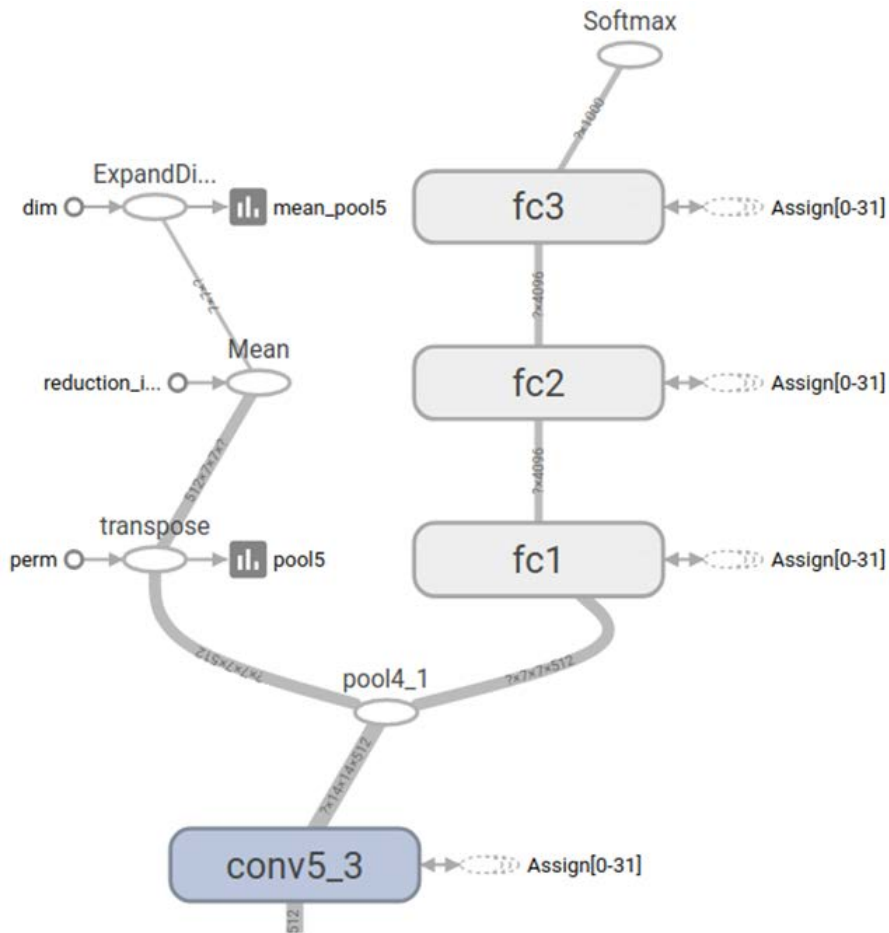


Figure 12.7 A small segment of the computation graph shown in TensorBoard for the VGG-16 neural network. The top-most node is the softmax operator used for classification. The three fully connected layers are labeled “fc1”, “fc2”, and “fc3”.

12.3 Ranking images

We will use the VGG-16 code in the previous section to obtain a vector representation of an image. That way, we can rank two images efficiently in the ranking neural network we designed in section 12.1.

Consider videos of cloth-folding, as shown in figure 12.8. We will process videos, frame-by-frame to rank the states of the images. That way, in a novel situation, the algorithm can understand whether the goal of cloth-folding has been reached.

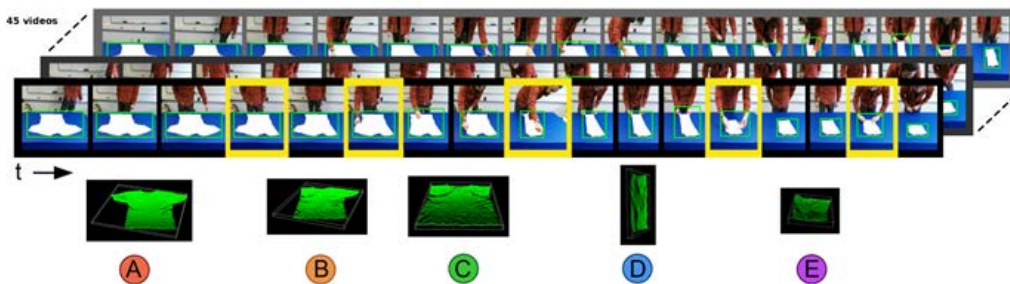


Figure 12.8 Videos of folding a shirt reveal how the cloth changes form through time. We can extract the first state and the last state of the shirt as our training data to learn a utility function to rank states. Final states of a cloth in each video should be ranked with a higher utility than those shirts near the beginning of the video.

First, download the cloth-folding dataset from <https://data.mendeley.com/datasets/c7y3hcrj7z/1>. Extract the zip. Keep note of where you extract it; we'll call that location DATASET_DIR in the code listings.

Open a new source file, and begin by importing the relevant libraries in Python as shown in figure 12.13.

Listing 12.11 Import libraries

```
import tensorflow as tf
import numpy as np
from vgg16 import vgg16
import glob, os
from scipy.misc import imread, imresize
```

For each video, we will remember the first and the last image. That way, we can train the ranking algorithm by assuming the last image is of a higher preference than the first image. In other words, the last state of cloth-folding brings you to a higher valued state than the first state of cloth-folding. Listing 12.12 shows an example of how to load the data into memory.

Listing 12.12 Prepare the training data

```
DATASET_DIR = os.path.join(os.path.expanduser('~'), 'res', 'cloth_folding_rgb_vids') // #A
NUM_VIDS = 45 // #B

def get_img_pair(video_id): // #C
    img_files = sorted(glob.glob(os.path.join(DATASET_DIR, video_id, '*.png')))
    start_img = img_files[0]
    end_img = img_files[-1]
    pair = []
    for image_file in [start_img, end_img]:
        img_original = imread(image_file)
        img_resized = imresize(img_original, (224, 224))
        pair.append(img_resized)
    return tuple(pair)

start_imgs = []
```



```

end_imgs= []
for vid_id in range(1, NUM_VIDS + 1):
    start_img, end_img = get_img_pair(str(vid_id))
    start_imgs.append(start_img)
    end_imgs.append(end_img)
print('Images of starting state {}'.format(np.shape(start_imgs)))
print('Images of ending state {}'.format(np.shape(end_imgs)))

```

```

#A Directory of downloaded files
#B Number of videos to load
#C Get starting and ending image of a video

```

Running listing 12.12 results in the following output:

```

Images of starting state (45, 224, 224, 3)
Images of ending state (45, 224, 224, 3)

```

Follow listing 12.13 to create an input placeholder for the image that we will be embedding.

Listing 12.13 Placeholders

```

imgs_plc = tf.placeholder(tf.float32, [None, 224, 224, 3])

```

Copy over the ranking neural network code from listings 12.3 – 12.7. We'll be reusing all that to rank images. Then let's prepare the session in listing 12.14.

Listing 12.14 Prepare the session

```

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

```

Next we will initialize the VGG-16 model by calling the constructor. Doing so, as shown in listing 12.15, will load all the model parameters from disk to memory.

Listing 12.15 Load the VGG-16 model

```

print('Loading model...')
vgg = vgg16(imgs_plc, 'vgg16_weights.npz', sess)
print('Done loading!')

```

Next, let's prepare training and testing data for the ranking neural network. As shown in listing 12.16, we will be feeding the VGG-16 model our images, and then we'll access a layer near the output (in this case `fc1`) to obtain the image embedding.

In the end, we will have 4096-dimensional embedding of our images. Because there are a total of 45 videos, we'll split some for training and some for testing:

- Train
 - Start frames size: (33, 4096)
 - End frames size: (33, 4096)
- Test
 - Start frames size: (12, 4096)
 - End frames size: (12, 4096)

Listing 12.16 Preparing data for ranking

```

start_imgs_embedded = sess.run(vgg.fc1, feed_dict={vgg.imgs: start_imgs})
end_imgs_embedded = sess.run(vgg.fc1, feed_dict={vgg.imgs: end_imgs})

idxs = np.random.choice(NUM_VIDS, NUM_VIDS, replace=False)
train_idx = idxs[0:int(NUM_VIDS * 0.75)]
test_idx = idxs[int(NUM_VIDS * 0.75):]

train_start_imgs = start_imgs_embedded[train_idx]
train_end_imgs = end_imgs_embedded[train_idx]
test_start_imgs = start_imgs_embedded[test_idx]
test_end_imgs = end_imgs_embedded[test_idx]

print('Train start imgs {}'.format(np.shape(train_start_imgs)))
print('Train end imgs {}'.format(np.shape(train_end_imgs)))
print('Test start imgs {}'.format(np.shape(test_start_imgs)))
print('Test end imgs {}'.format(np.shape(test_end_imgs)))

```

With our training data ready for ranking, let's run the `train_op` an epoch number of times. After training the network, run the model on the test data to evaluate our results.

Listing 12.17 Train the ranking network

```

train_y1 = np.expand_dims(np.zeros(np.shape(train_start_imgs)[0]), axis=1)
train_y2 = np.expand_dims(np.ones(np.shape(train_end_imgs)[0]), axis=1)
for epoch in range(100):
    for i in range(np.shape(train_start_imgs)[0]):
        _, cost_val = sess.run([train_op, loss],
                               feed_dict={x1: train_start_imgs[i:i+1,:],
                                             x2: train_end_imgs[i:i+1,:],
                                             dropout_keep_prob: 0.5})
    print('{} {}'.format(epoch, cost_val))
    s1_val, s2_val = sess.run([s1, s2], feed_dict={x1: test_start_imgs,
                                                  x2: test_end_imgs,
                                                  dropout_keep_prob: 1})
    print('Accuracy: {}'.format(100 * np.mean(s1_val < s2_val)))

```

Notice how the accuracy approaches 100% over time. That means our ranking model learns that the images that occur at the end of the video are more favorable than the images that occur near the beginning.

Just out of curiosity, let's see the utility over time frame-by-frame of a single video, as shown in figure 12.9. The code to reproduce the figure shown below requires loading all the images in a video, as outlined in listing 12.18.

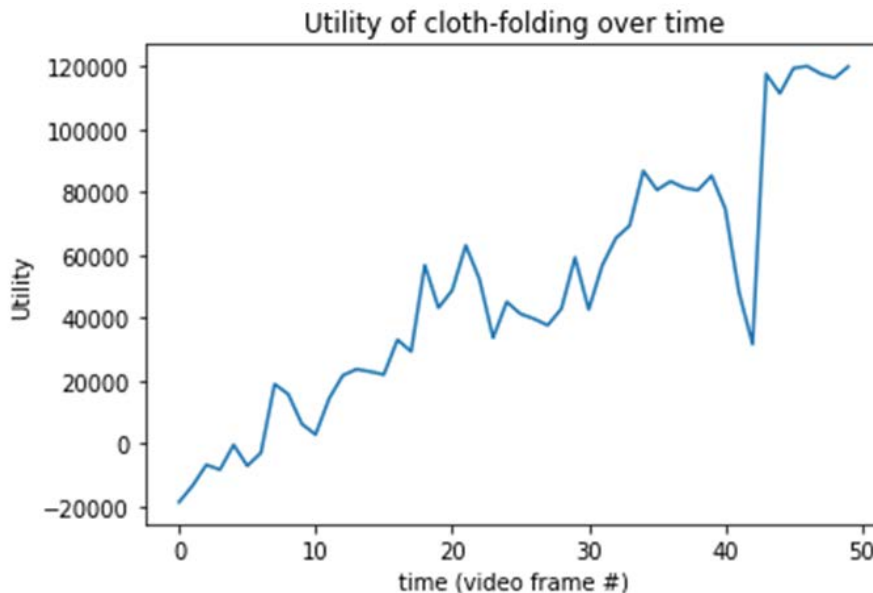


Figure 12.9 The utility increases over time, indicating the goal is being accomplished. The utility of the cloth near the beginning of the video is near 0, but it dramatically increases to 120000 units by the end.

Listing 12.18 Prepare image sequences from video

```
def get_img_seq(video_id):
    img_files = sorted(glob.glob(os.path.join(DATASET_DIR, video_id, '*.png')))
    imgs = []
    for image_file in img_files:
        img_original = imread(image_file)
        img_resized = imresize(img_original, (224, 224))
        imgs.append(img_resized)
    return imgs

imgs = get_img_seq('1')
```

We can use our VGG-16 model to embed the images, and then run the ranking network to compute the scores (listing 12.19).

Listing 12.19 Compute utility of images

```
imgs_embedded = sess.run(vgg.fc1, feed_dict={vgg.imgs: imgs})
scores = sess.run([s1], feed_dict={x1: imgs_embedded,
                                   dropout_keep_prob: 1})
```

Let's visualize our results to reproduce figure 12.7. See listing 12.20.

Listing 12.20

```
from matplotlib import pyplot as plt
```

```
plt.figure()
plt.title('Utility of cloth-folding over time')
plt.xlabel('time (video frame #)')
plt.ylabel('Utility')
plt.plot(scores[-1])
```

12.4 Summary

In this chapter, you learned the following:

- How to rank states, by representing objects as vectors and learning a utility function over such vectors.
- Images contain redundant data, so we used the VGG-16 neural network to reduce the dimensionality of our data so that we can use the ranking network with real-world images.
- How to visualize the utility of images over time in a video, to verify that the video demonstration increases utility of the cloth.

You've finishing your TensorFlow journey! The 12 chapters of this book each approached ML in different angles; however, the whole book unifies concepts to muster these skills:

- Formulate an arbitrary real-world problem into a machine learning framework.
- Learn basics of many machine learning problems
- Use TensorFlow to solve these machine learning problems
- Visualize a machine learning algorithm and speak the lingo

12.5 What's next?

Because the concepts taught in this book are timeless, the code listings should be too. To ensure the most up-to- date library calls and syntax, I actively manage a GitHub repository at <https://github.com/BinRoot/TensorFlow-Book>. Please feel free to join the community there and file bugs or send me pull requests.

REMINDER TensorFlow is rapidly being developed so more functionality will become available all the time!

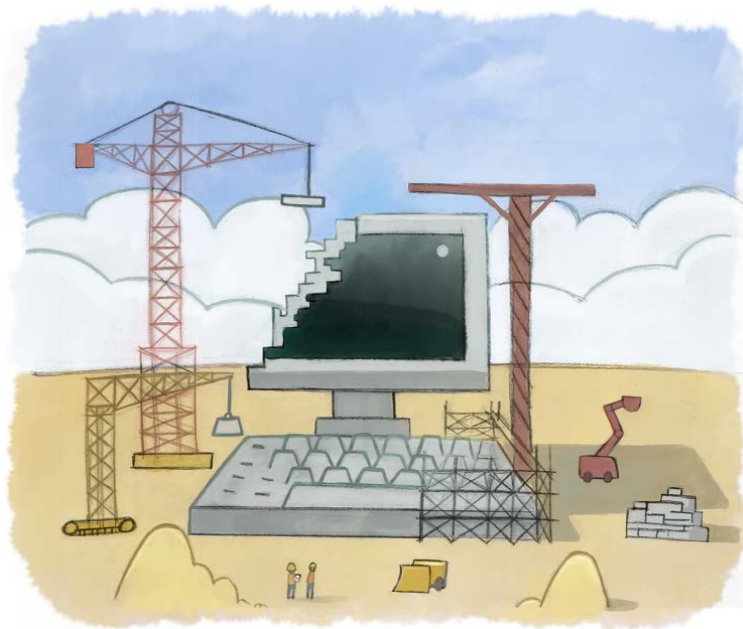
If you're thirsty for more TensorFlow tutorials, I know exactly what might interest you:

1. Reinforcement Learning (RL) - an in-depth series of blog-posts on using RL in TensorFlow:
<https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0>
2. Natural Language Processing (NLP) - an essential TensorFlow guide to modern neural network architectures in NLP:
http://www.thushv.com/natural_language_processing/word2vec-part-1-nlp-with-deep-learning-with-tensorflow-skip-gram/

3. Generative Adversarial Networks (GAN) – an introductory study of generative vs. discriminative models in machine learning (using TensorFlow):
<http://blog.aylien.com/introduction-generative-adversarial-networks-code-tensorflow/>
4. Web tool – tinker with a simple neural network to visualize the flow of data:
<http://playground.tensorflow.org>
5. Video lectures – basic introduction and hand-on demos on using TensorFlow:
<https://cloud.google.com/blog/big-data/2017/01/learn-tensorflow-and-deep-learning-without-a-phd>
6. Open source projects – follow along to the most recently updated TensorFlow projects on GitHub:
<https://github.com/search?o=desc&q=tensorflow&s=updated&type=Repositories>

A

Installation



This appendix covers

- Installing TensorFlow
- Installing Matplotlib

You can install TensorFlow in a couple of ways. This book assumes you'll be using Python 3 for every chapter unless otherwise stated. The code listings abide by TensorFlow v1.0, but the accompanying source code on GitHub will always be up-to-date to the latest version (<https://github.com/BinRoot/TensorFlow-Book/>). In this appendix, we will cover one of these installation methods that works on all platforms, including Windows. If you're familiar with UNIX based systems (like Linux or OSX), feel free to use one of the installation approaches on the official documentation: https://www.tensorflow.org/get_started/os_setup.html.

Without further ado, 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 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, you're almost good to go.

The next step is to check if your processor can support virtualization. On windows 8 or higher, you can open the Task Manager (Ctrl + Shift + Esc) and click 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>).

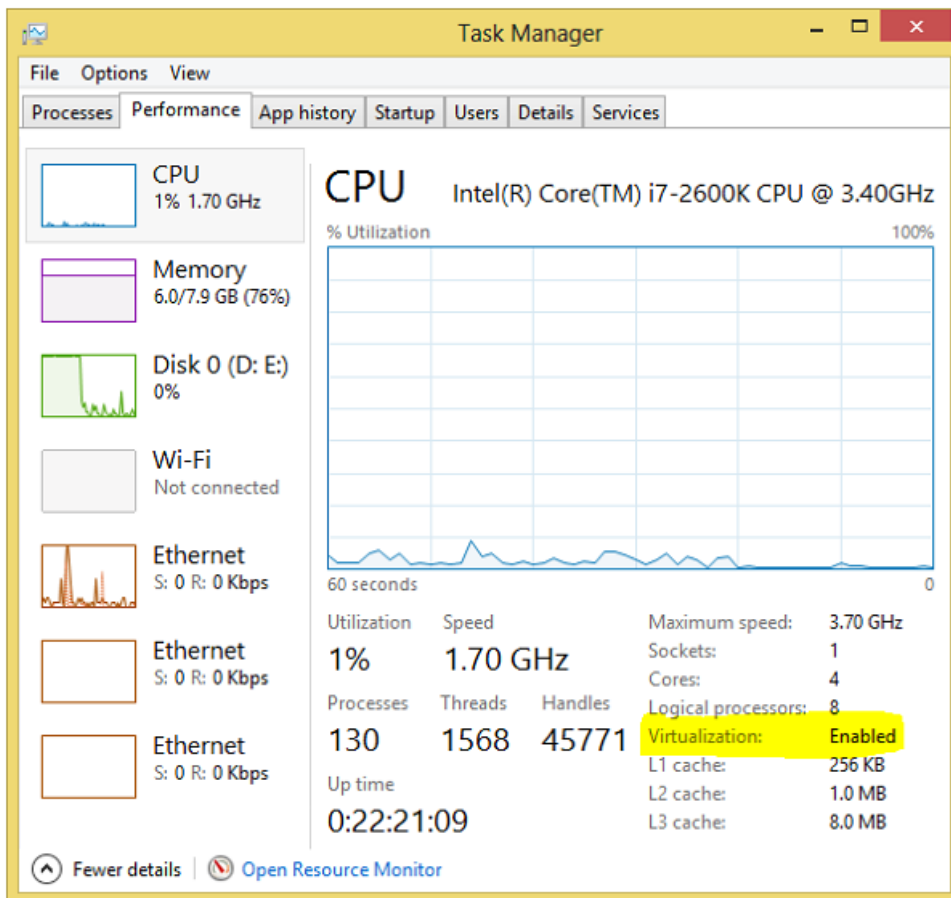


Figure A.1 Ensure your 64-bit computer has virtualization enabled.

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 several 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

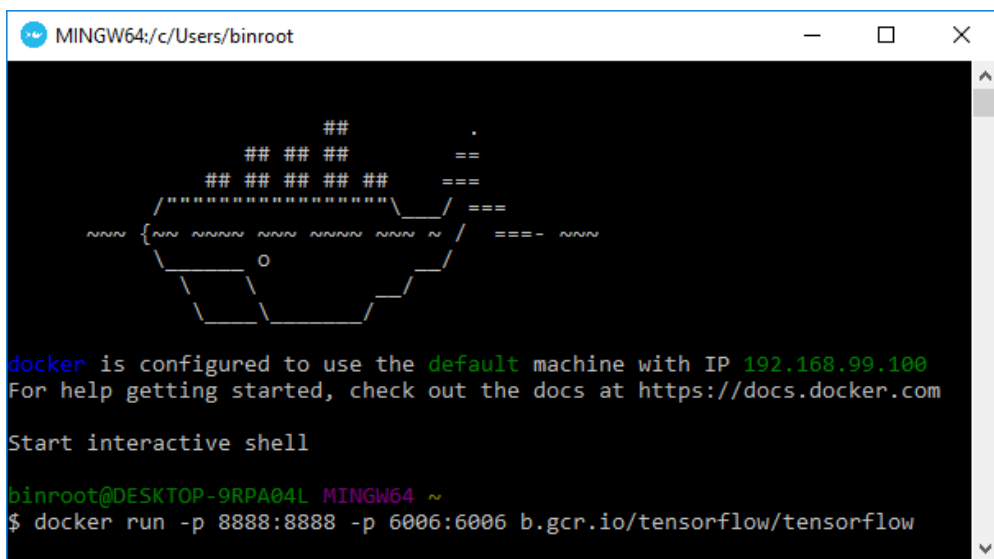
Docker works on OSX 10.8 “Mountain Lion” or newer. Install the Docker Toolbox from <https://www.docker.com/products/docker-toolbox>. After installation, open the “Docker Quickstart Terminal” from the Applications folder or the Launchpad.

A.1.4 How to user Docker

Run the Docker Quickstart Terminal.

Next, launch the TensorFlow binary image using the following command in the docker terminal, as shown in Figure A.2.

```
$ docker run -p 8888:8888 -p 6006:6006 b.gcr.io/tensorflow/tensorflow
```



```

MINGW64; c/Users/binroot
      ##
     ## ## ##
    ## ## ## ## ##
   {-----}
  NNN {NN NNNN NNN NNNN NNN N} NNN
      O
     / \
    /   \
   /     \
  /       \
 /         \
/           \

docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

Start interactive shell

binroot@DESKTOP-9RPA04L MINGW64 ~
$ docker run -p 8888:8888 -p 6006:6006 b.gcr.io/tensorflow/tensorflow

```

Figure A.2 Run the official TensorFlow

TensorFlow will now be accessible from a Jupyter notebook via a local IP address. The IP can be found using the `docker-machine ip` command, as shown in figure A.3.

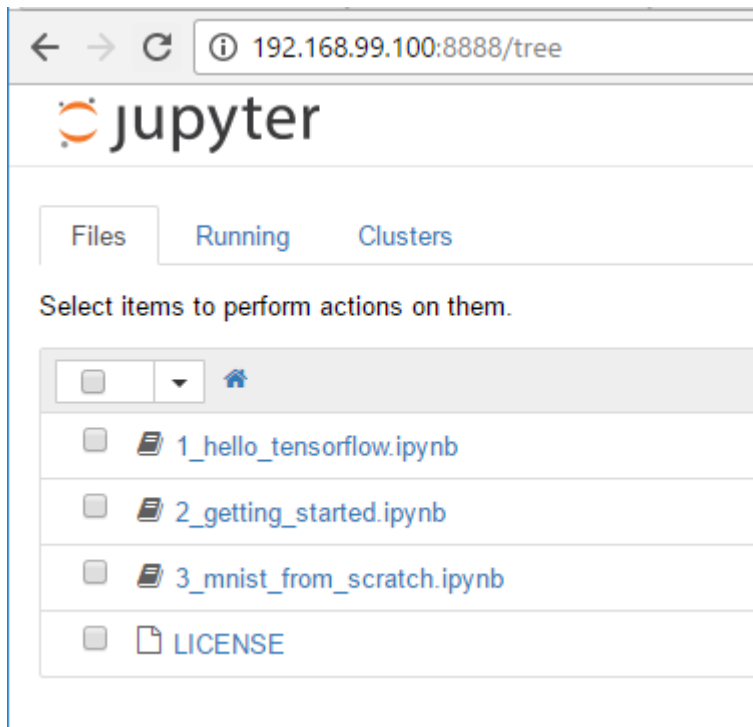


Figure A.4 We can interact with TensorFlow through a Python interface called Jupyter.

You can press `Ctrl+C` or close the terminal window to stop running Jupyter notebook. To re-run it, just follow the steps in this section again.

If you run into the error message shown in figure A.5, it means Docker is already using an application on that port.

```
binroot@DESKTOP-9RPA04L MINGW64 ~
$ docker run -p 8888:8888 -p 6006:6006 b.gcr.io/tensorflow/tensorflow
C:\Program Files\ Docker Toolbox\docker.exe: Error response from daemon:
driver failed programming external connectivity on endpoint tender_allen
(ab6dcf2455a5704f8f2911ac53ea946deb3ed939864c30e8fe867c2f5c88a63d): Bin
d for 0.0.0.0:8888 failed: port is already allocated.
```

Figure A.5 A possible error message from running the TensorFlow container

To resolve this issue, you can either switch the port or quit the intruding Docker containers. Figure A.6 shows how to list all containers using `docker ps` and then kill the container using `docker kill`.

```
binroot@DESKTOP-9RPA04L MINGW64 ~  
$ docker ps  
CONTAINER ID          IMAGE  
62904e0a4489         b.gcr.io/tensorflow/tensorflow  
  
binroot@DESKTOP-9RPA04L MINGW64 ~  
$ docker kill 62904e0a4489  
62904e0a4489
```

Figure A.6 How to list and kill a Docker container to get rid of the error message in figure A.5

A.2 Installing Matplotlib

Matplotlib is a cross-platform Python library for plotting 2D visualizations of data. Generally, 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>.