Fundamentals of Deep Learning

Designing Next-Generation Machine Intelligence Algorithms

Nikhil Buduma

with contributions by Nicholas Locascio

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

Fundamentals of Deep Learning

by Nikhil Buduma and Nicholas Lacascio

Printed in the United States of America

June 2017: First Edition

Revision History for the First Edition
2017-05-25: First Release

978-1-491-92561-4

[TI]

Contents

Preface

With the reinvigoration of neural networks in the 2000s, deep learning has become an extremely active area of research that is paving the way for modern machine learning. This book uses exposition and examples to help you understand major concepts in this complicated field. Large companies such as Google, Microsoft, and Facebook have taken notice and are actively growing in-house deep learning teams. For the rest of us, deep learning is still a pretty complex and difficult subject to grasp. Research papers are filled to the brim with jargon, and scattered online tutorials do little to help build a strong intuition for why and how deep learning practitioners approach problems. Our goal is to bridge this gap.

Prerequisites and Objectives

This booked is aimed an audience with a basic operating understanding of calculus, matrices, and Python programming. Approaching this material without this background is possible, but likely to be more challenging. Background in linear algebra may also be helpful in navigating certain sections of mathematical exposition.

By the end of the book, we hope that our readers will be left with an intuition for how to approach problems using deep learning, the historical context for modern deep learning approaches, and a familiarity with implementing deep learning algorithms using the TensorFlow open source library.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/darksigma/Fundamentals-of-Deep-Learning-Book*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Fundamentals of Deep Learning* by Nikhil Buduma and Nicholas Locascio (O'Reilly). Copyright 2017 Nikhil Buduma and Nicholas Locascio, 978-1-491-92561-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1

The Neural Network

Building Intelligent Machines

The brain is the most incredible organ in the human body. It dictates the way we perceive every sight, sound, smell, taste, and touch. It enables us to store memories, experience emotions, and even dream. Without it, we would be primitive organisms, incapable of anything other than the simplest of reflexes. The brain is, inherently, what makes us intelligent.

The infant brain only weighs a single pound, but somehow it solves problems that even our biggest, most powerful supercomputers find impossible. Within a matter of months after birth, infants can recognize the faces of their parents, discern discrete objects from their backgrounds, and even tell apart voices. Within a year, they've already developed an intuition for natural physics, can track objects even when they become partially or completely blocked, and can associate sounds with specific meanings. And by early childhood, they have a sophisticated understanding of grammar and thousands of words in their vocabularies.[1]

For decades, we've dreamed of building intelligent machines with brains like ours—robotic assistants to clean our homes, cars that drive themselves, microscopes that automatically detect diseases. But building these artificially intelligent machines requires us to solve some of the most complex computational problems we have ever grappled with; problems that our brains can already solve in a manner of microseconds. To tackle these problems, we'll have to develop a radically different way of programming a computer using techniques largely developed over the past decade. This

1 Kuhn, Deanna, et al. *Handbook of Child Psychology. Vol. 2, Cognition, Perception, and Language*. Wiley, 1998.

is an extremely active field of artificial computer intelligence often referred to as *deep learning*.

The Limits of Traditional Computer Programs

Why exactly are certain problems so difficult for computers to solve? Well, it turns out that traditional computer programs are designed to be very good at two things: 1) performing arithmetic really fast and 2) explicitly following a list of instructions. So if you want to do some heavy financial number crunching, you're in luck. Traditional computer programs can do the trick. But let's say we want to do something slightly more interesting, like write a program to automatically read someone's handwriting. Figure 1-1 will serve as a starting point.

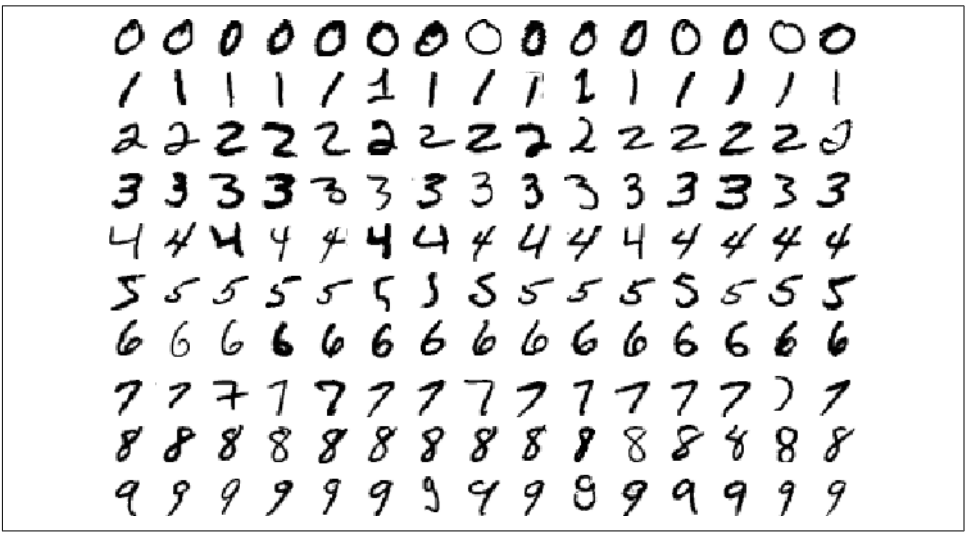


Figure 1-1. Image from MNIST handwritten digit dataset[2]

Although every digit in Figure 1-1 is written in a slightly different way, we can easily recognize every digit in the first row as a zero, every digit in the second row as a one, etc. Let's try to write a computer program to crack this task. What rules could we use to tell one digit from another?

Well, we can start simple! For example, we might state that we have a zero if our image only has a single, closed loop. All the examples in Figure 1-1 seem to fit this bill, but this isn't really a sufficient condition. What if someone doesn't perfectly close

2 Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition" *Proceedings of the IEEE*, 86(11):2278-2324, November 1998.

the loop on their zero? And, as in Figure 1-2, how do you distinguish a messy zero from a six?

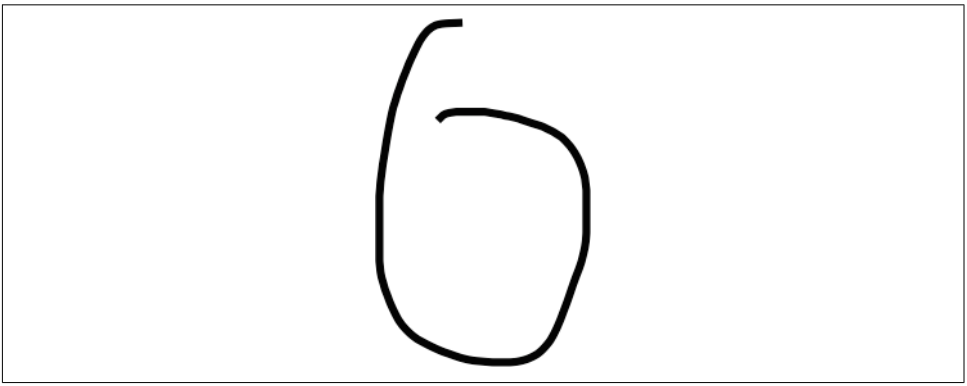


Figure 1-2. A zero that's algorithmically difficult to distinguish from a six

You could potentially establish some sort of cutoff for the distance between the starting point of the loop and the ending point, but it's not exactly clear where we should be drawing the line. But this dilemma is only the beginning of our worries. How do we distinguish between threes and fives? Or between fours and nines? We can add more and more rules, or *features*, through careful observation and months of trial and error, but it's quite clear that this isn't going to be an easy process.

Many other classes of problems fall into this same category: object recognition, speech comprehension, automated translation, etc. We don't know what program to write because we don't know how it's done by our brains. And even if we did know how to do it, the program might be horrendously complicated.

The Mechanics of Machine Learning

To tackle these classes of problems, we'll have to use a very different kind of approach. A lot of the things we learn in school growing up have a lot in common with traditional computer programs. We learn how to multiply numbers, solve equations, and take derivatives by internalizing a set of instructions. But the things we learn at an extremely early age, the things we find most natural, are learned by example, not by formula.

For instance, when we were two years old, our parents didn't teach us how to recognize a dog by measuring the shape of its nose or the contours of its body. We learned to recognize a dog by being shown multiple examples and being corrected when we made the wrong guess. In other words, when we were born, our brains provided us with a model that described how we would be able to see the world. As we grew up, that model would take in our sensory inputs and make a guess about what we were

experiencing. If that guess was confirmed by our parents, our model would be reinforced. If our parents said we were wrong, we'd modify our model to incorporate this new information. Over our lifetime, our model becomes more and more accurate as we assimilate more and more examples. Obviously all of this happens subconsciously, without us even realizing it, but we can use this to our advantage nonetheless.

Deep learning is a subset of a more general field of artificial intelligence called *machine learning*, which is predicated on this idea of learning from example. In machine learning, instead of teaching a computer a massive list of rules to solve the problem, we give it a *model* with which it can evaluate examples, and a small set of instructions to modify the model when it makes a mistake. We expect that, over time, a well-suited model would be able to solve the problem extremely accurately.

Let's be a little bit more rigorous about what this means so we can formulate this idea mathematically. Let's define our model to be a function $h(\mathbf{x}, \theta)$. The input $\mathbf{x}$ is an example expressed in vector form. For example, if $\mathbf{x}$ were a grayscale image, the vector's components would be pixel intensities at each position, as shown in Figure 1-3.

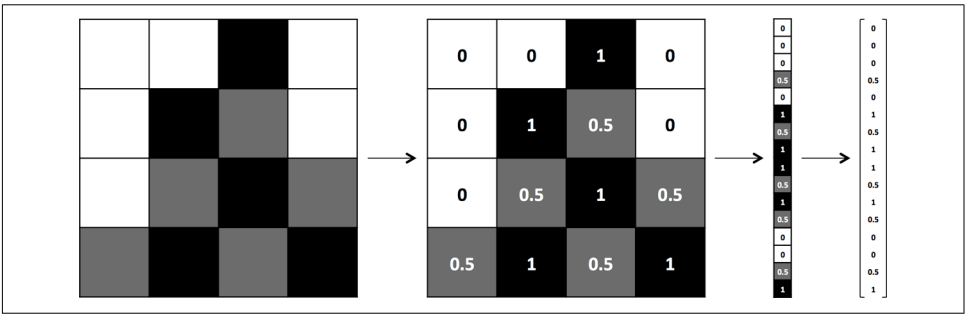


Figure 1-3. The process of vectorizing an image for a machine learning algorithm

The input θ is a vector of the parameters that our model uses. Our machine learning program tries to perfect the values of these parameters as it is exposed to more and more examples. We'll see this in action and in more detail in Chapter 2.

To develop a more intuitive understanding for machine learning models, let's walk through a quick example. Let's say we wanted to determine how to predict exam performance based on the number of hours of sleep we get and the number of hours we study the previous day. We collect a lot of data, and for each data point $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$, we record the number of hours of sleep we got (x_1), the number of hours we spent studying (x_2), and whether we performed above or below the class average. Our goal, then, might be to learn a model $h(\mathbf{x}, \theta)$ with parameter vector $\theta = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 \end{bmatrix}^T$ such that:

$$h(\mathbf{x}, \theta) = \begin{cases} -1 & \text{if } \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 < 0 \\ 1 & \text{if } \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 \geq 0 \end{cases}$$

In other words, we guess that the blueprint for our model $h(\mathbf{x}, \theta)$ is as described above (geometrically, this particular blueprint describes a linear classifier that divides the coordinate plane into two halves). Then, we want to learn a parameter vector θ such that our model makes the right predictions (-1 if we perform below average, and 1 otherwise) given an input example $\mathbf{x}$. This model is called a linear *perceptron*, and it's a model that's been used since the 1950s.[3] Let's assume our data is as shown in Figure 1-4.

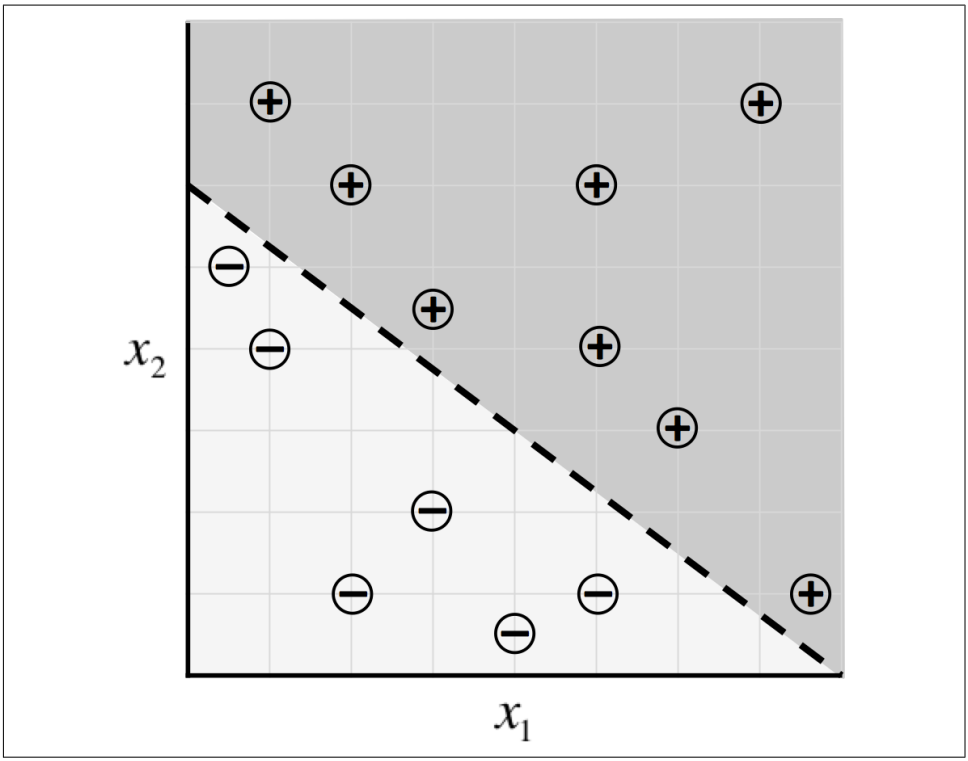


Figure 1-4. Sample data for our exam predictor algorithm and a potential classifier

3 Rosenblatt, Frank. "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review* 65.6 (1958): 386.

Then it turns out, by selecting $\theta = [-24 \;\; 3 \;\; 4]^T$, our machine learning model makes the correct prediction on every data point:

$$h(\mathbf{x}, \theta) = \begin{cases} -1 & \text{if } 3x_1 + 4x_2 - 24 < 0 \\ 1 & \text{if } 3x_1 + 4x_2 - 24 \geq 0 \end{cases}$$

An optimal parameter vector θ positions the classifier so that we make as many correct predictions as possible. In most cases, there are many (or even infinitely many) possible choices for θ that are optimal. Fortunately for us, most of the time these alternatives are so close to one another that the difference is negligible. If this is not the case, we may want to collect more data to narrow our choice of θ.

While the setup seems reasonable, there are still some pretty significant questions that remain. First off, how do we even come up with an optimal value for the parameter vector θ in the first place? Solving this problem requires a technique commonly known as *optimization*. An optimizer aims to maximize the performance of a machine learning model by iteratively tweaking its parameters until the error is minimized. We'll begin to tackle this question of learning parameter vectors in more detail in Chapter 2, when we describe the process of *gradient descent*.[4] In later chapters, we'll try to find ways to make this process even more efficient.

Second, it's quite clear that this particular model (the linear perceptron model) is quite limited in the relationships it can learn. For example, the distributions of data shown in Figure 1-5 cannot be described well by a linear perceptron.

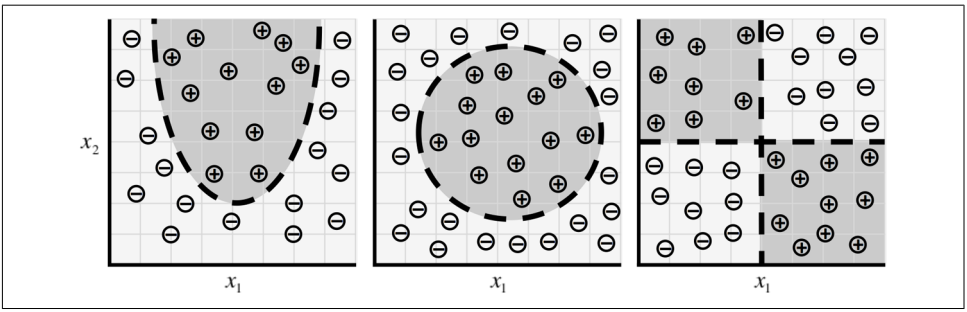


Figure 1-5. As our data takes on more complex forms, we need more complex models to describe them

But these situations are only the tip of the iceberg. As we move on to much more complex problems, such as object recognition and text analysis, our data becomes extremely high dimensional, and the relationships we want to capture become highly

4 Bubeck, Sébastien. "Convex optimization: Algorithms and complexity." *Foundations and Trends® in Machine Learning*. 8.3-4 (2015): 231-357.

nonlinear. To accommodate this complexity, recent research in machine learning has attempted to build models that resemble the structures utilized by our brains. It's essentially this body of research, commonly referred to as *deep learning*, that has had spectacular success in tackling problems in computer vision and natural language processing. These algorithms not only far surpass other kinds of machine learning algorithms, but also rival (or even exceed!) the accuracies achieved by humans.

The Neuron

The foundational unit of the human brain is the neuron. A tiny piece of the brain, about the size of grain of rice, contains over 10,000 neurons, each of which forms an average of 6,000 connections with other neurons.[5] It's this massive biological network that enables us to experience the world around us. Our goal in this section will be to use this natural structure to build machine learning models that solve problems in an analogous way.

At its core, the neuron is optimized to receive information from other neurons, process this information in a unique way, and send its result to other cells. This process is summarized in Figure 1-6. The neuron receives its inputs along antennae-like structures called *dendrites*. Each of these incoming connections is dynamically strengthened or weakened based on how often it is used (this is how we learn new concepts!), and it's the strength of each connection that determines the contribution of the input to the neuron's output. After being weighted by the strength of their respective connections, the inputs are summed together in the *cell body*. This sum is then transformed into a new signal that's propagated along the cell's *axon* and sent off to other neurons.

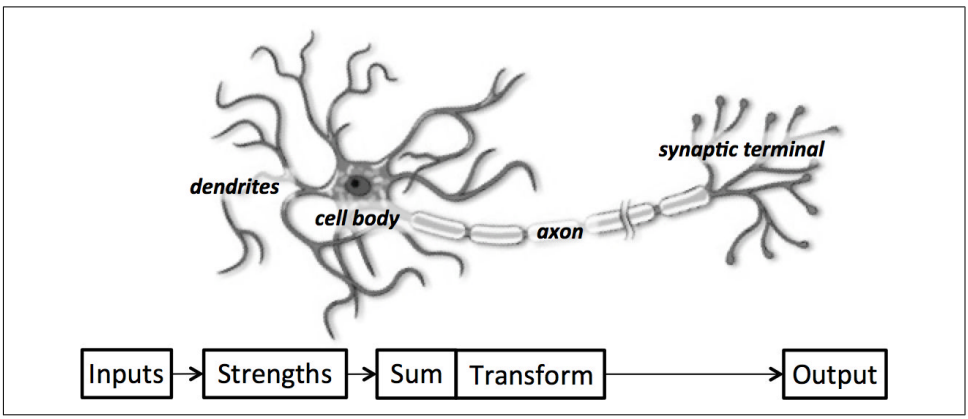


Figure 1-6. A functional description of a biological neuron's structure

5 Restak, Richard M. and David Grubin. *The Secret Life of the Brain*. Joseph Henry Press, 2001.

We can translate this functional understanding of the neurons in our brain into an artificial model that we can represent on our computer. Such a model is described in Figure 1-7, leveraging the approach first pioneered in 1943 by Warren S. McCulloch and Walter H. Pitts.[6] Just as in biological neurons, our artificial neuron takes in some number of inputs, $x_1, x_2, \ldots, x_n$, each of which is multiplied by a specific weight, $w_1, w_2, \ldots, w_n$. These weighted inputs are, as before, summed together to produce the *logit* of the neuron, $z = \sum_{i=0}^{n} w_i x_i$. In many cases, the logit also includes a *bias*, which is a constant (not shown in the figure). The logit is then passed through a function f to produce the output $y = f(z)$. This output can be transmitted to other neurons.

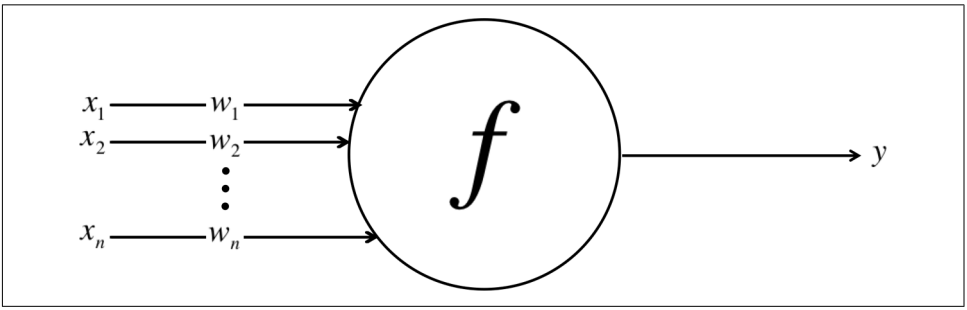


Figure 1-7. Schematic for a neuron in an artificial neural net

We'll conclude our mathematical discussion of the artificial neuron by re-expressing its functionality in vector form. Let's reformulate the inputs as a vector $\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]$ and the weights of the neuron as $\mathbf{w} = [w_1\ w_2\ \ldots\ w_n]$. Then we can re-express the output of the neuron as $y = f(\mathbf{x} \cdot \mathbf{w} + b)$, where b is the bias term. In other words, we can compute the output by performing the dot product of the input and weight vectors, adding in the bias term to produce the logit, and then applying the transformation function. While this seems like a trivial reformulation, thinking about neurons as a series of vector manipulations will be crucial to how we implement them in software later in this book.

Expressing Linear Perceptrons as Neurons

In "The Mechanics of Machine Learning" on page 3, we talked about using machine learning models to capture the relationship between success on exams and time spent studying and sleeping. To tackle this problem, we constructed a linear perceptron classifier that divided the Cartesian coordinate plane into two halves:

6 McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." *The Bulletin of Mathematical Biophysics*. 5.4 (1943): 115-133.

$$h(\mathbf{x}, \theta) = \begin{cases} -1 & \text{if } 3x_1 + 4x_2 - 24 < 0 \\ 1 & \text{if } 3x_1 + 4x_2 - 24 \geq 0 \end{cases}$$

As shown in Figure 1-4, this is an optimal choice for θ because it correctly classifies every sample in our dataset. Here, we show that our model h is easily using a neuron. Consider the neuron depicted in Figure 1-8. The neuron has two inputs, a bias, and uses the function:

$$f(z) = \begin{cases} -1 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

It's very easy to show that our linear perceptron and the neuronal model are perfectly equivalent. And in general, it's quite simple to show that singular neurons are strictly more expressive than linear perceptrons. In other words, every linear perceptron can be expressed as a single neuron, but single neurons can also express models that cannot be expressed by any linear perceptron.

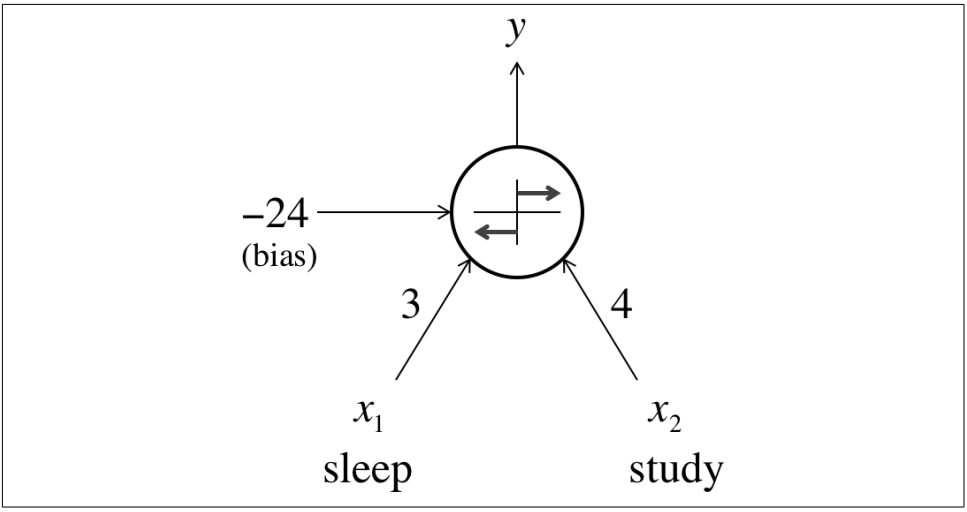


Figure 1-8. Expressing our exam performance perceptron as a neuron

Feed-Forward Neural Networks

Although single neurons are more powerful than linear perceptrons, they're not nearly expressive enough to solve complicated learning problems. There's a reason our brain is made of more than one neuron. For example, it is impossible for a single neuron to differentiate handwritten digits. So to tackle much more complicated tasks, we'll have to take our machine learning model even further.

The neurons in the human brain are organized in layers. In fact, the human cerebral cortex (the structure responsible for most of human intelligence) is made up of six

layers.[7] Information flows from one layer to another until sensory input is converted into conceptual understanding. For example, the bottommost layer of the visual cortex receives raw visual data from the eyes. This information is processed by each layer and passed on to the next until, in the sixth layer, we conclude whether we are looking at a cat, or a soda can, or an airplane. Figure 1-9 shows a more simplified version of these layers.

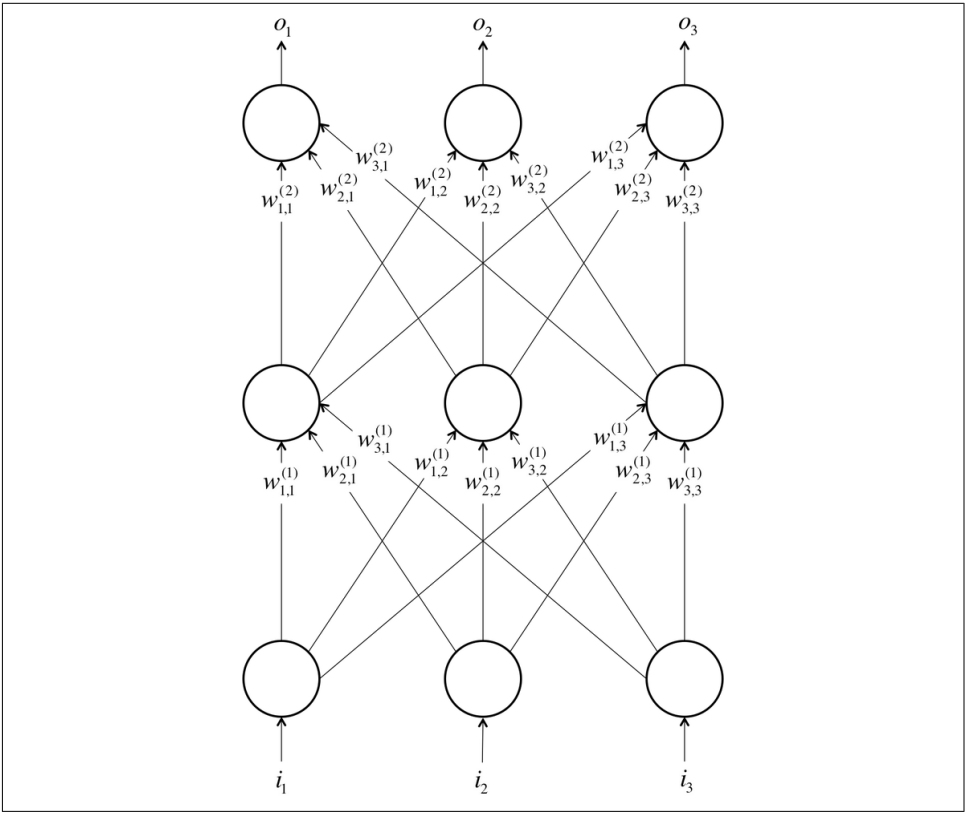


Figure 1-9. A simple example of a feed-forward neural network with three layers (input, one hidden, and output) and three neurons per layer

Borrowing from these concepts, we can construct an *artificial neural network*. A neural network comes about when we start hooking up neurons to each other, the input data, and to the output nodes, which correspond to the network's answer to a learning problem. Figure 1-9 demonstrates a simple example of an artificial neural network, similar to the architecture described in McCulloch and Pitt's work in 1943. The

7 Mountcastle, Vernon B. "Modality and topographic properties of single neurons of cat's somatic sensory cortex." *Journal of Neurophysiology* 20.4 (1957): 408-434.

bottom layer of the network pulls in the input data. The top layer of neurons (output nodes) computes our final answer. The middle layer(s) of neurons are called the *hidden layers,* and we let $w_{i,j}^{(k)}$ be the weight of the connection between the i^{th} neuron in the k^{th} layer with the j^{th} neuron in the $k+1^{st}$ layer. These weights constitute our parameter vector, θ, and just as before, our ability to solve problems with neural networks depends on finding the optimal values to plug into θ.

We note that in this example, connections only traverse from a lower layer to a higher layer. There are no connections between neurons in the same layer, and there are no connections that transmit data from a higher layer to a lower layer. These neural networks are called *feed-forward* networks, and we start by discussing these networks because they are the simplest to analyze. We present this analysis (specifically, the process of selecting the optimal values for the weights) in Chapter 2. More complicated connectivities will be addressed in later chapters.

In the final sections, we'll discuss the major types of layers that are utilized in feed-forward neural networks. But before we proceed, here's a couple of important notes to keep in mind:

1. As we mentioned, the layers of neurons that lie sandwiched between the first layer of neurons (input layer) and the last layer of neurons (output layer) are called the hidden layers. This is where most of the magic is happening when the neural net tries to solve problems. Whereas (as in the handwritten digit example) we would previously have to spend a lot of time identifying useful features, the hidden layers automate this process for us. Oftentimes, taking a look at the activities of hidden layers can tell you a lot about the features the network has automatically learned to extract from the data.
2. Although in this example every layer has the same number of neurons, this is neither necessary nor recommended. More often than not, hidden layers have fewer neurons than the input layer to force the network to learn compressed representations of the original input. For example, while our eyes obtain raw pixel values from our surroundings, our brain thinks in terms of edges and contours. This is because the hidden layers of biological neurons in our brain force us to come up with better representations for everything we perceive.
3. It is not required that every neuron has its output connected to the inputs of all neurons in the next layer. In fact, selecting which neurons to connect to which other neurons in the next layer is an art that comes from experience. We'll discuss this issue in more depth as we work through various examples of neural networks.
4. The inputs and outputs are *vectorized* representations. For example, you might imagine a neural network where the inputs are the individual pixel RGB values in an image represented as a vector (refer to Figure 1-3). The last layer might have two neurons that correspond to the answer to our problem: $[1, 0]$ if the image

contains a dog, $[0, 1]$ if the image contains a cat, $[1, 1]$ if it contains both, and $[0, 0]$ if it contains neither.

We'll also observe that, similarly to our reformulation for the neuron, we can also mathematically express a neural network as a series of vector and matrix operations. Let's consider the input to the i^{th} layer of the network to be a vector $\mathbf{x} = [x_1\ x_2\ ...\ x_n]$. We'd like to find the vector $\mathbf{y} = [y_1\ y_2\ ...\ y_m]$ produced by propagating the input through the neurons. We can express this as a simple matrix multiply if we construct a weight matrix $\mathbf{W}$ of size $n \times m$ and a bias vector of size m. In this matrix, each column corresponds to a neuron, where the j^{th} element of the column corresponds to the weight of the connection pulling in the j^{th} element of the input. In other words, $\mathbf{y} = f(\mathbf{W}^T\mathbf{x} + \mathbf{b})$, where the transformation function is applied to the vector elementwise. This reformulation will become all the more critical as we begin to implement these networks in software.

Linear Neurons and Their Limitations

Most neuron types are defined by the function f they apply to their logit z. Let's first consider layers of neurons that use a linear function in the form of $f(z) = az + b$. For example, a neuron that attempts to estimate a cost of a meal in a fast-food restaurant would use a linear neuron where $a = 1$ and $b = 0$. In other words, using $f(z) = z$ and weights equal to the price of each item, the linear neuron in Figure 1-10 would take in some ordered triple of servings of burgers, fries, and sodas and output the price of the combination.

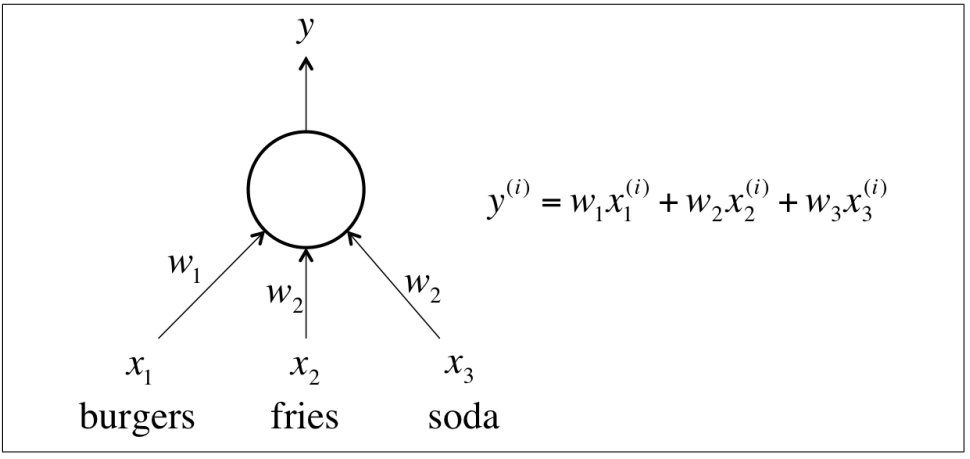


Figure 1-10. An example of a linear neuron

Linear neurons are easy to compute with, but they run into serious limitations. In fact, it can be shown that any feed-forward neural network consisting of only linear

neurons can be expressed as a network with no hidden layers. This is problematic because, as we discussed before, hidden layers are what enable us to learn important features from the input data. In other words, in order to learn complex relationships, we need to use neurons that employ some sort of nonlinearity.

Sigmoid, Tanh, and ReLU Neurons

There are three major types of neurons that are used in practice that introduce nonlinearities in their computations. The first of these is the *sigmoid neuron*, which uses the function:

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

Intuitively, this means that when the logit is very small, the output of a logistic neuron is very close to 0. When the logit is very large, the output of the logistic neuron is close to 1. In-between these two extremes, the neuron assumes an S-shape, as shown in Figure 1-11.

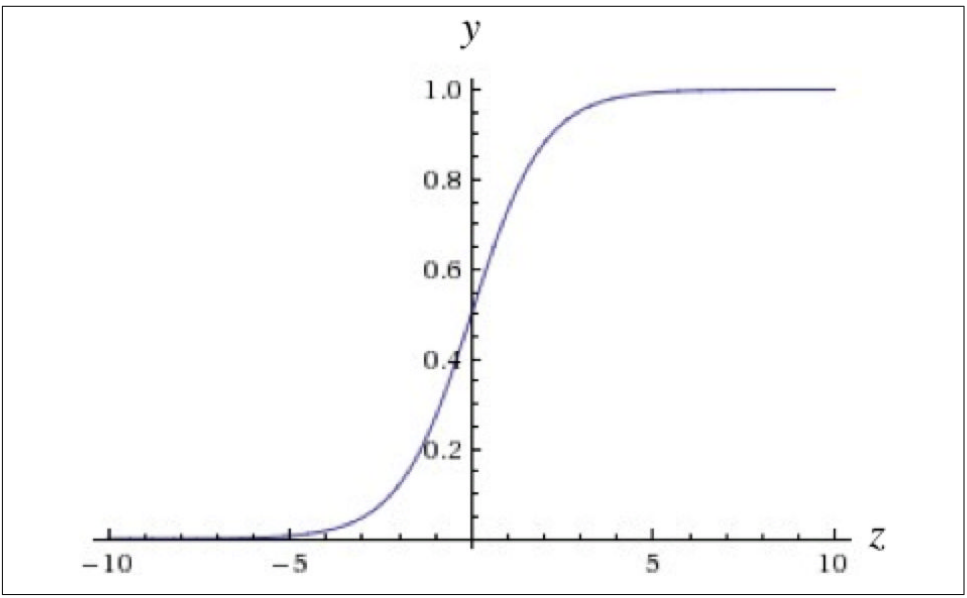


Figure 1-11. The output of a sigmoid neuron as z varies

Tanh neurons use a similar kind of S-shaped nonlinearity, but instead of ranging from 0 to 1, the output of tanh neurons range from −1 to 1. As one would expect, they use $f(z) = \tanh(z)$. The resulting relationship between the output y and the logit z is described by Figure 1-12. When S-shaped nonlinearities are used, the tanh neuron is often preferred over the sigmoid neuron because it is zero-centered.

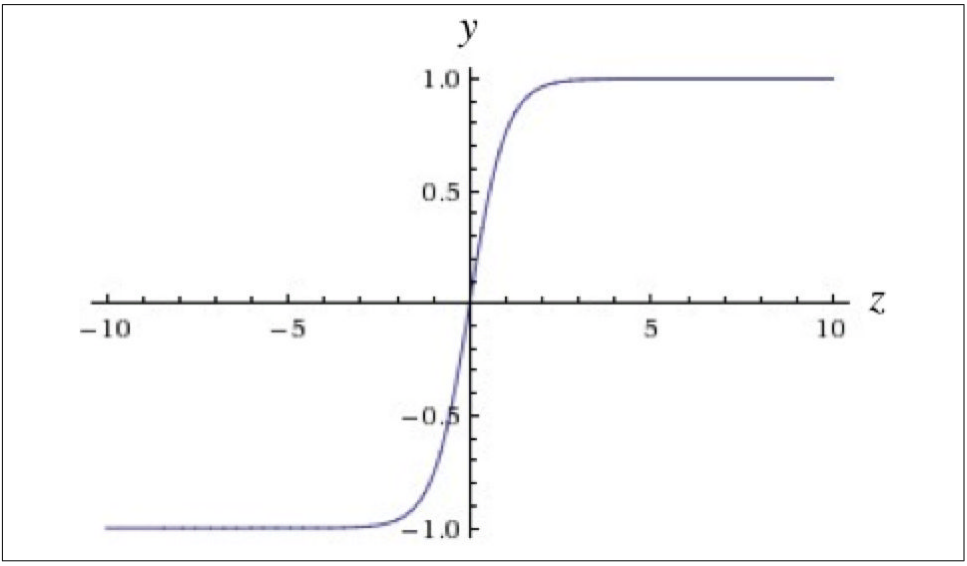


Figure 1-12. The output of a tanh neuron as z varies

A different kind of nonlinearity is used by the *restricted linear unit (ReLU) neuron*. It uses the function $f(z) = \max(0, z)$, resulting in a characteristic hockey-stick-shaped response, as shown in Figure 1-13.

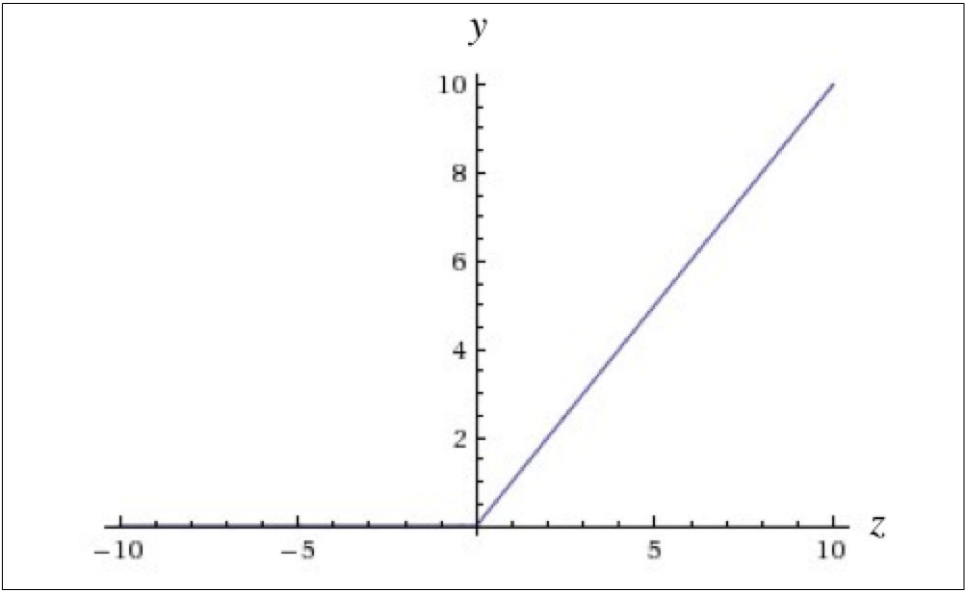


Figure 1-13. The output of a ReLU neuron as z varies

The ReLU has recently become the neuron of choice for many tasks (especially in computer vision) for a number of reasons, despite some drawbacks.[8] We'll discuss these reasons in Chapter 5, as well as strategies to combat the potential pitfalls.

Softmax Output Layers

Oftentimes, we want our output vector to be a probability distribution over a set of mutually exclusive labels. For example, let's say we want to build a neural network to recognize handwritten digits from the MNIST dataset. Each label (0 through 9) is mutually exclusive, but it's unlikely that we will be able to recognize digits with 100% confidence. Using a probability distribution gives us a better idea of how confident we are in our predictions. As a result, the desired output vector is of the form below, where $\Sigma_{i=0}^{9} p_i = 1$:

$$\begin{bmatrix} p_0 & p_1 & p_2 & p_3 & \cdots & p_9 \end{bmatrix}$$

This is achieved by using a special output layer called a *softmax layer*. Unlike in other kinds of layers, the output of a neuron in a softmax layer depends on the outputs of all the other neurons in its layer. This is because we require the sum of all the outputs to be equal to 1. Letting z_i be the logit of the i^{th} softmax neuron, we can achieve this normalization by setting its output to:

$$y_i = \frac{e^{z_i}}{\Sigma_j e^{z_j}}$$

A strong prediction would have a single entry in the vector close to 1, while the remaining entries were close to 0. A weak prediction would have multiple possible labels that are more or less equally likely.

Looking Forward

In this chapter, we've built a basic intuition for machine learning and neural networks. We've talked about the basic structure of a neuron, how feed-forward neural networks work, and the importance of nonlinearity in tackling complex learning problems. In the next chapter, we will begin to build the mathematical background necessary to train a neural network to solve problems. Specifically, we will talk about finding optimal parameter vectors, best practices while training neural networks, and major challenges. In future chapters, we will take these foundational ideas to build more specialized neural architectures.

---

8 Nair, Vinod, and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines" *Proceedings of the 27th International Conference on Machine Learning* (ICML-10), 2010.

CHAPTER 2

Training Feed-Forward Neural Networks

The Fast-Food Problem

We're beginning to understand how we can tackle some interesting problems using deep learning, but one big question still remains: how exactly do we figure out what the parameter vectors (the weights for all of the connections in our neural network) should be? This is accomplished by a process commonly referred to as *training* (see Figure 2-1). During training, we show the neural net a large number of training examples and iteratively modify the weights to minimize the errors we make on the training examples. After enough examples, we expect that our neural network will be quite effective at solving the task it's been trained to do.

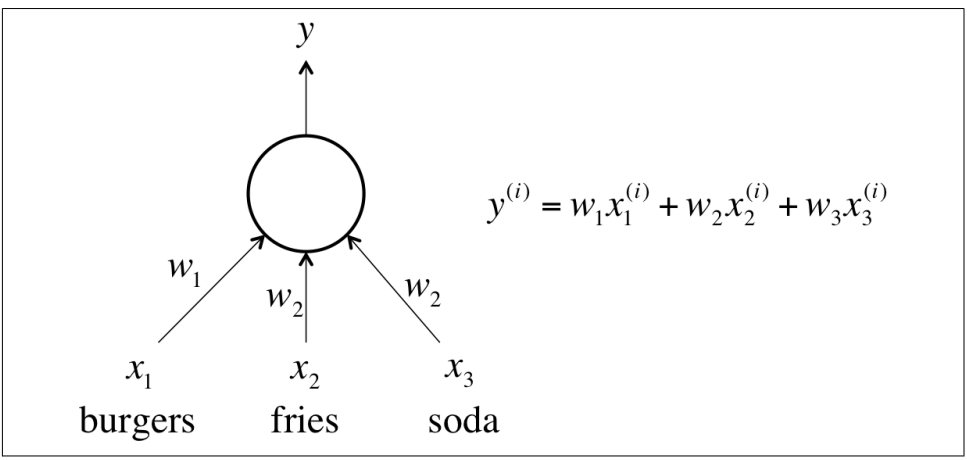


Figure 2-1. This is the neuron we want to train for the fast-food problem

Let's continue with the example we mentioned in the previous chapter involving a linear neuron. As a brief review, every single day, we purchase a restaurant meal consisting of burgers, fries, and sodas. We buy some number of servings for each item. We want to be able to predict how much a meal is going to cost us, but the items don't have price tags. The only thing the cashier will tell us is the total price of the meal. We want to train a single linear neuron to solve this problem. How do we do it?

One idea is to be intelligent about picking our training cases. For one meal we could buy only a single serving of burgers, for another we could only buy a single serving of fries, and then for our last meal we could buy a single serving of soda. In general, intelligently selecting training examples is a very good idea. There's lots of research that shows that by engineering a clever training set, you can make your neural network a lot more effective. The issue with using this approach alone is that in real situations, it rarely ever gets you close to the solution. For example, there's no clear analog of this strategy in image recognition. It's just not a practical solution.

Instead, we try to motivate a solution that works well in general. Let's say we have a large set of training examples. Then we can calculate what the neural network will output on the i^{th} training example using the simple formula in the diagram. We want to train the neuron so that we pick the optimal weights possible—the weights that minimize the errors we make on the training examples. In this case, let's say we want to minimize the square error over all of the training examples that we encounter. More formally, if we know that $t^{(i)}$ is the true answer for the i^{th} training example and $y^{(i)}$ is the value computed by the neural network, we want to minimize the value of the error function E:

$$E = \frac{1}{2}\Sigma_i\left(t^{(i)} - y^{(i)}\right)^2$$

The squared error is zero when our model makes a perfectly correct prediction on every training example. Moreover, the closer E is to 0, the better our model is. As a result, our goal will be to select our parameter vector θ (the values for all the weights in our model) such that E is as close to 0 as possible.

Now at this point you might be wondering why we need to bother ourselves with error functions when we can treat this problem as a system of equations. After all, we have a bunch of unknowns (weights) and we have a set of equations (one for each training example). That would automatically give us an error of 0, assuming that we have a consistent set of training examples.

That's a smart observation, but the insight unfortunately doesn't generalize well. Remember that although we're using a linear neuron here, linear neurons aren't used very much in practice because they're constrained in what they can learn. And the moment we start using nonlinear neurons like the sigmoidal, tanh, or ReLU neurons

we talked about at the end of the previous chapter, we can no longer set up a system of equations! Clearly we need a better strategy to tackle the training process.

Gradient Descent

Let's visualize how we might minimize the squared error over all of the training examples by simplifying the problem. Let's say our linear neuron only has two inputs (and thus only two weights, w_1 and w_2). Then we can imagine a three-dimensional space where the horizontal dimensions correspond to the weights w_1 and w_2, and the vertical dimension corresponds to the value of the error function E. In this space, points in the horizontal plane correspond to different settings of the weights, and the height at those points corresponds to the incurred error. If we consider the errors we make over all possible weights, we get a surface in this three-dimensional space, in particular, a quadratic bowl as shown in Figure 2-2.

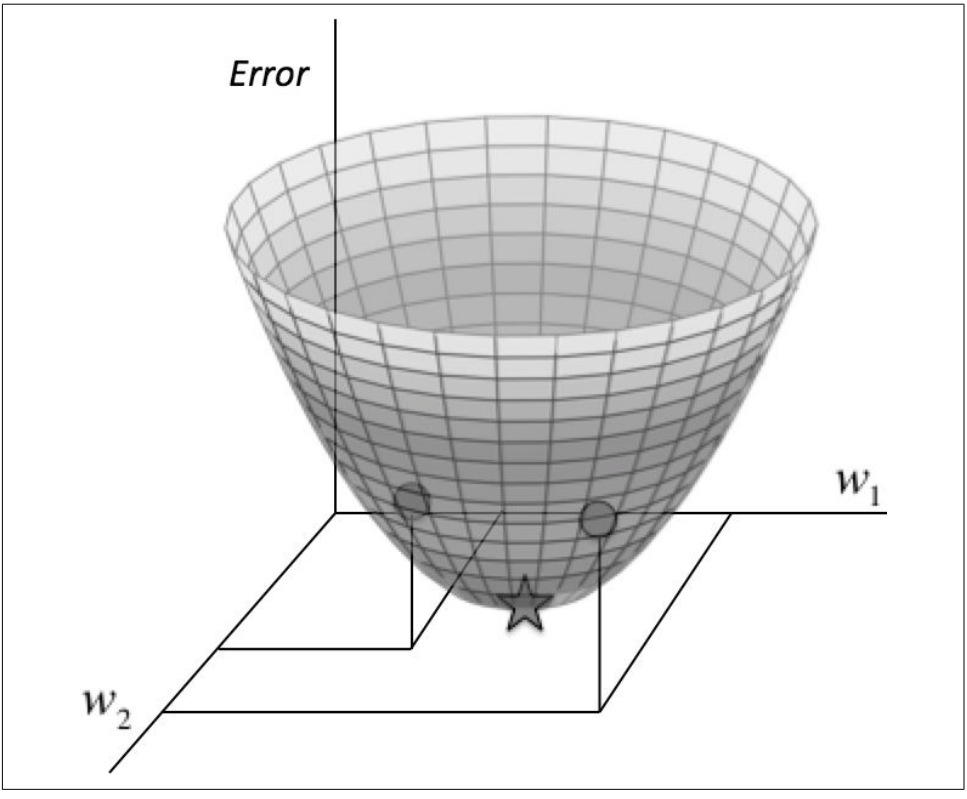


Figure 2-2. The quadratic error surface for a linear neuron

We can also conveniently visualize this surface as a set of elliptical contours, where the minimum error is at the center of the ellipses. In this setup, we are working in a two-dimensional plane where the dimensions correspond to the two weights. Contours correspond to settings of w_1 and w_2 that evaluate to the same value of E. The closer the contours are to each other, the steeper the slope. In fact, it turns out that the direction of the steepest descent is always perpendicular to the contours. This direction is expressed as a vector known as the *gradient*.

Now we can develop a high-level strategy for how to find the values of the weights that minimizes the error function. Suppose we randomly initialize the weights of our network so we find ourselves somewhere on the horizontal plane. By evaluating the gradient at our current position, we can find the direction of steepest descent, and we can take a step in that direction. Then we'll find ourselves at a new position that's closer to the minimum than we were before. We can reevaluate the direction of steepest descent by taking the gradient at this new position and taking a step in this new direction. It's easy to see that, as shown in Figure 2-3, following this strategy will eventually get us to the point of minimum error. This algorithm is known as *gradient descent*, and we'll use it to tackle the problem of training individual neurons and the more general challenge of training entire networks.[1]

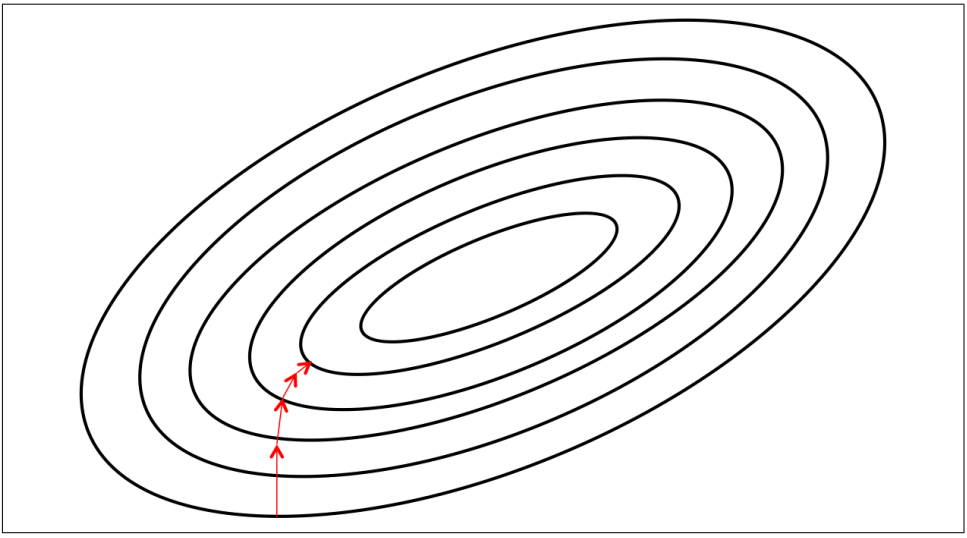


Figure 2-3. Visualizing the error surface as a set of contours

1 Rosenbloom, P. "The method of steepest descent." *Proceedings of Symposia in Applied Mathematics*. Vol. 6. 1956.

The Delta Rule and Learning Rates

Before we derive the exact algorithm for training our fast-food neuron, a quick note on *hyperparameters*. In addition to the weight parameters defined in our neural network, learning algorithms also require a couple of additional parameters to carry out the training process. One of these so-called hyperparameters is the *learning rate*.

In practice, at each step of moving perpendicular to the contour, we need to determine how far we want to walk before recalculating our new direction. This distance needs to depend on the steepness of the surface. Why? The closer we are to the minimum, the shorter we want to step forward. We know we are close to the minimum, because the surface is a lot flatter, so we can use the steepness as an indicator of how close we are to the minimum. However, if our error surface is rather mellow, training can potentially take a large amount of time. As a result, we often multiply the gradient by a factor ϵ, the learning rate. Picking the learning rate is a hard problem (Figure 2-4). As we just discussed, if we pick a learning rate that's too small, we risk taking too long during the training process. But if we pick a learning rate that's too big, we'll mostly likely start diverging away from the minimum. In Chapter 3, we'll learn about various optimization techniques that utilize adaptive learning rates to automate the process of selecting learning rates.

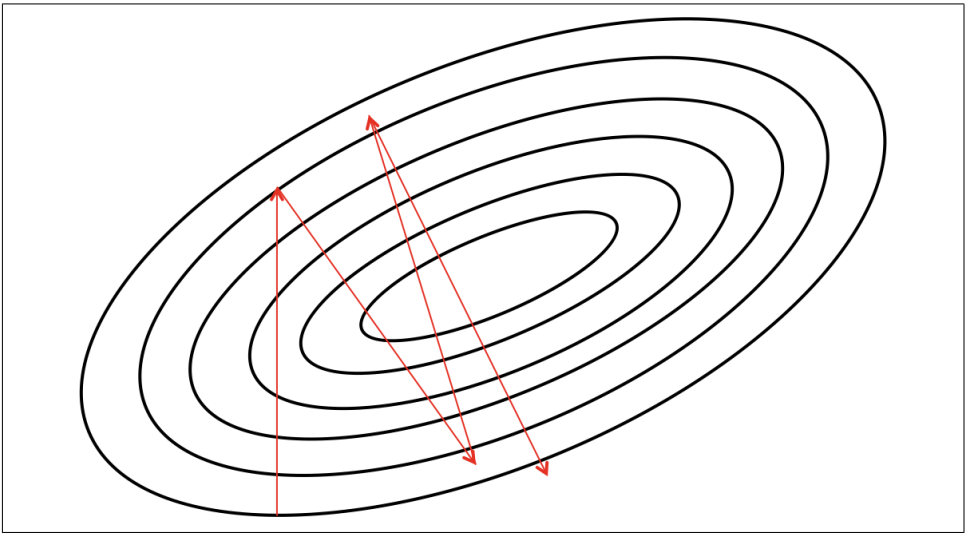


Figure 2-4. Convergence is difficult when our learning rate is too large

Now, we are finally ready to derive the *delta rule* for training our linear neuron. In order to calculate how to change each weight, we evaluate the gradient, which is essentially the partial derivative of the error function with respect to each of the weights. In other words, we want:

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}$$

$$= -\epsilon \frac{\partial}{\partial w_k}\left(\frac{1}{2}\Sigma_i \left(t^{(i)} - y^{(i)}\right)^2\right)$$

$$= \Sigma_i \epsilon \left(t^{(i)} - y^{(i)}\right) \frac{\partial y_i}{\partial w_k}$$

$$= \Sigma_i \epsilon x_k^{(i)} \left(t^{(i)} - y^{(i)}\right)$$

Applying this method of changing the weights at every iteration, we are finally able to utilize gradient descent.

Gradient Descent with Sigmoidal Neurons

In this section and the next, we will deal with training neurons and neural networks that utilize nonlinearities. We use the sigmoidal neuron as a model, and leave the derivations for other nonlinear neurons as an exercise for the reader. For simplicity, we assume that the neurons do not use a bias term, although our analysis easily extends to this case. We merely need to assume that the bias is a weight on an incoming connection whose input value is always one.

Let's recall the mechanism by which logistic neurons compute their output value from their inputs:

$$z = \Sigma_k w_k x_k$$

$$y = \frac{1}{1 + e^{-z}}$$

The neuron computes the weighted sum of its inputs, the logit z. It then feeds its logit into the input function to compute y, its final output. Fortunately for us, these functions have very nice derivatives, which makes learning easy! For learning, we want to compute the gradient of the error function with respect to the weights. To do so, we start by taking the derivative of the logit with respect to the inputs and the weights:

$$\frac{\partial z}{\partial w_k} = x_k$$

$$\frac{\partial z}{\partial x_k} = w_k$$

Also, quite surprisingly, the derivative of the output with respect to the logit is quite simple if you express it in terms of the output:

$$\frac{dy}{dz} = \frac{e^{-z}}{\left(1 + e^{-z}\right)^2}$$

$$= \frac{1}{1+e^{-z}} \frac{e^{-z}}{1+e^{-z}}$$

$$= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$$

$$= y(1-y)$$

We then use the chain rule to get the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz}\frac{\partial z}{\partial w_k} = x_k y(1-y)$$

Putting all of this together, we can now compute the derivative of the error function with respect to each weight:

$$\frac{\partial E}{\partial w_k} = \Sigma_i \frac{\partial E}{\partial y^{(i)}} \frac{\partial y^{(i)}}{\partial w_k} = -\Sigma_i x_k^{(i)} y^{(i)} \left(1 - y^{(i)}\right)\left(t^{(i)} - y^{(i)}\right)$$

Thus, the final rule for modifying the weights becomes:

$$\Delta w_k = \Sigma_i \epsilon x_k^{(i)} y^{(i)} \left(1 - y^{(i)}\right)\left(t^{(i)} - y^{(i)}\right)$$

As you may notice, the new modification rule is just like the delta rule, except with extra multiplicative terms included to account for the logistic component of the sigmoidal neuron.

The Backpropagation Algorithm

Now we're finally ready to tackle the problem of training multilayer neural networks (instead of just single neurons). To accomplish this task, we'll use an approach known as *backpropagation*, pioneered by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams in 1986.[2] So what's the idea behind backpropagation? We don't know what the hidden units ought to be doing, but what we can do is compute how fast the error changes as we change a hidden activity. From there, we can figure out how fast the error changes when we change the weight of an individual connection. Essentially, we'll be trying to find the path of steepest descent! The only catch is that we're going to be working in an extremely high-dimensional space. We start by calculating the error derivatives with respect to a single training example.

Each hidden unit can affect many output units. Thus, we'll have to combine many separate effects on the error in an informative way. Our strategy will be one of dynamic programming. Once we have the error derivatives for one layer of hidden

2 Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive Modeling* 5.3 (1988): 1.

units, we'll use them to compute the error derivatives for the activities of the layer below. And once we find the error derivatives for the activities of the hidden units, it's quite easy to get the error derivatives for the weights leading into a hidden unit. We'll redefine some notation for ease of discussion and refer to the Figure 2-5.

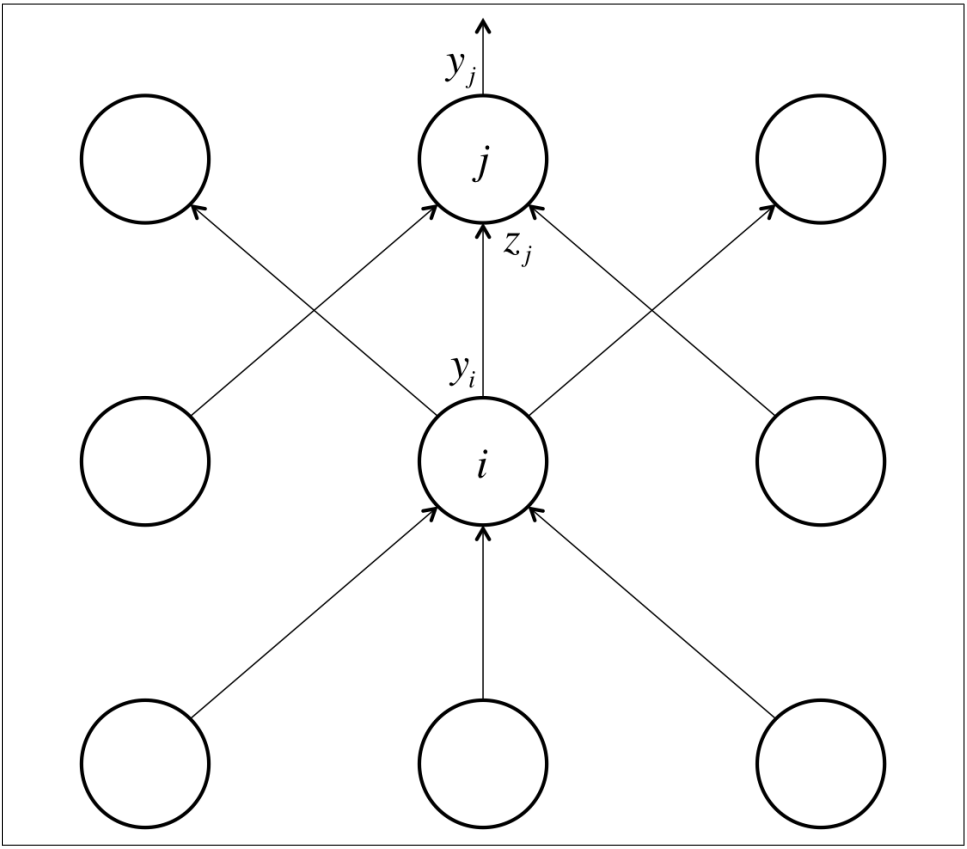


Figure 2-5. Reference diagram for the derivation of the backpropagation algorithm

The subscript we use will refer to the layer of the neuron. The symbol y will refer to the activity of a neuron, as usual. Similarly, the symbol z will refer to the logit of the neuron. We start by taking a look at the base case of the dynamic programming problem. Specifically, we calculate the error function derivatives at the output layer:

$$E = \frac{1}{2}\Sigma_{j \in output}\left(t_j - y_j\right)^2 \Rightarrow \frac{\partial E}{\partial y_j} = -\left(t_j - y_j\right)$$

Now we tackle the inductive step. Let's presume we have the error derivatives for layer j. We now aim to calculate the error derivatives for the layer below it, layer i. To do so, we must accumulate information about how the output of a neuron in layer i affects the logits of every neuron in layer j. This can be done as follows, using

the fact that the partial derivative of the logit with respect to the incoming output data from the layer beneath is merely the weight of the connection w_{ij}:

$$\frac{\partial E}{\partial y_i} = \Sigma_j \frac{\partial E}{\partial z_j} \frac{dz_j}{dy_i} = \Sigma_j w_{ij} \frac{\partial E}{\partial z_j}$$

Furthermore, we observe the following:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{dy_j}{dz_j} = y_j\left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

Combining these two together, we can finally express the error derivatives of layer *i* in terms of the error derivatives of layer *j*:

$$\frac{\partial E}{\partial y_i} = \Sigma_j w_{ij} y_j\left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

Then once we've gone through the whole dynamic programming routine, having filled up the table appropriately with all of our partial derivatives (of the error function with respect to the hidden unit activities), we can then determine how the error changes with respect to the weights. This gives us how to modify the weights after each training example:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i y_j\left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

Finally, to complete the algorithm, just as before, we merely sum up the partial derivatives over all the training examples in our dataset. This gives us the following modification formula:

$$\Delta w_{ij} = -\Sigma_{k \in dataset} \epsilon y_i^{(k)} y_j^{(k)}\left(1 - y_j^{(k)}\right) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$

This completes our description of the backpropagation algorithm!

Stochastic and Minibatch Gradient Descent

In the algorithms we've described in "The Backpropagation Algorithm" on page 23, we've been using a version of gradient descent known as *batch gradient descent*. The idea behind batch gradient descent is that we use our entire dataset to compute the error surface and then follow the gradient to take the path of steepest descent. For a simple quadratic error surface, this works quite well. But in most cases, our error surface may be a lot more complicated. Let's consider the scenario in Figure 2-6 for illustration.

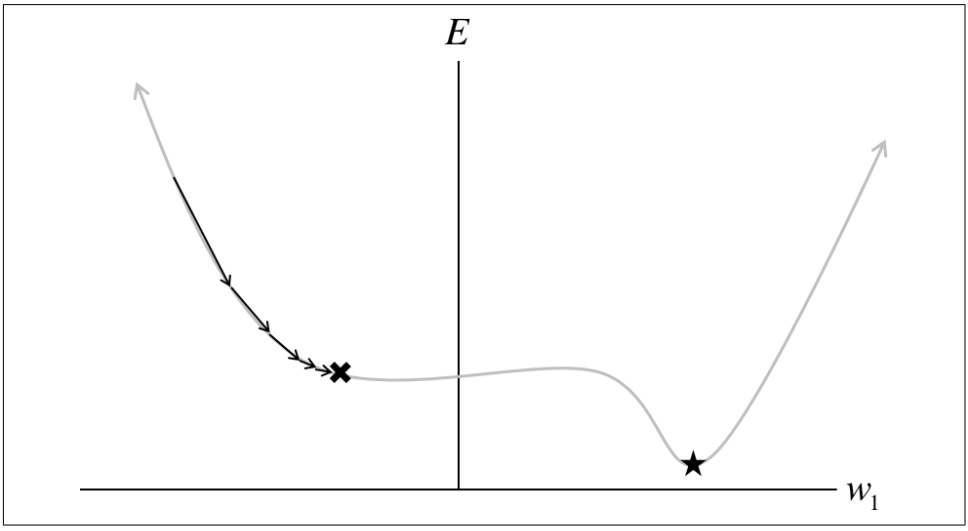


Figure 2-6. Batch gradient descent is sensitive to saddle points, which can lead to premature convergence

We only have a single weight, and we use random initialization and batch gradient descent to find its optimal setting. The error surface, however, has a flat region (also known as saddle point in high-dimensional spaces), and if we get unlucky, we might find ourselves getting stuck while performing gradient descent.

Another potential approach is *stochastic gradient descent* (SGD), where at each iteration, our error surface is estimated only with respect to a single example. This approach is illustrated by Figure 2-7, where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.

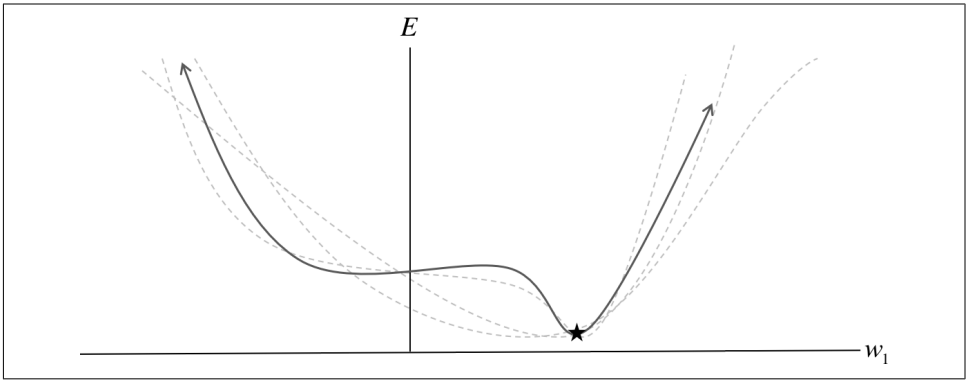


Figure 2-7. The stochastic error surface fluctuates with respect to the batch error surface, enabling saddle point avoidance

The major pitfall of stochastic gradient descent, however, is that looking at the error incurred one example at a time may not be a good enough approximation of the error surface. This, in turn, could potentially make gradient descent take a significant amount of time. One way to combat this problem is using *mini-batch gradient descent*. In mini-batch gradient descent, at every iteration, we compute the error surface with respect to some subset of the total dataset (instead of just a single example). This subset is called a *minibatch*, and in addition to the learning rate, minibatch size is another hyperparameter. Minibatches strike a balance between the efficiency of batch gradient descent and the local-minima avoidance afforded by stochastic gradient descent. In the context of backpropagation, our weight update step becomes:

$$\Delta w_{ij} = -\sum_{k \in minibatch} \epsilon y_i^{(k)} y_j^{(k)} \left(1 - y_j^{(k)}\right) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$

This is identical to what we derived in the previous section, but instead of summing over all the examples in the dataset, we sum over the examples in the current minibatch.

Test Sets, Validation Sets, and Overfitting

One of the major issues with artificial neural networks is that the models are quite complicated. For example, let's consider a neural network that's pulling data from an image from the MNIST database (28 x 28 pixels), feeds into two hidden layers with 30 neurons, and finally reaches a softmax layer of 10 neurons. The total number of parameters in the network is nearly 25,000. This can be quite problematic, and to understand why, let's consider a new toy example, illustrated in Figure 2-8.

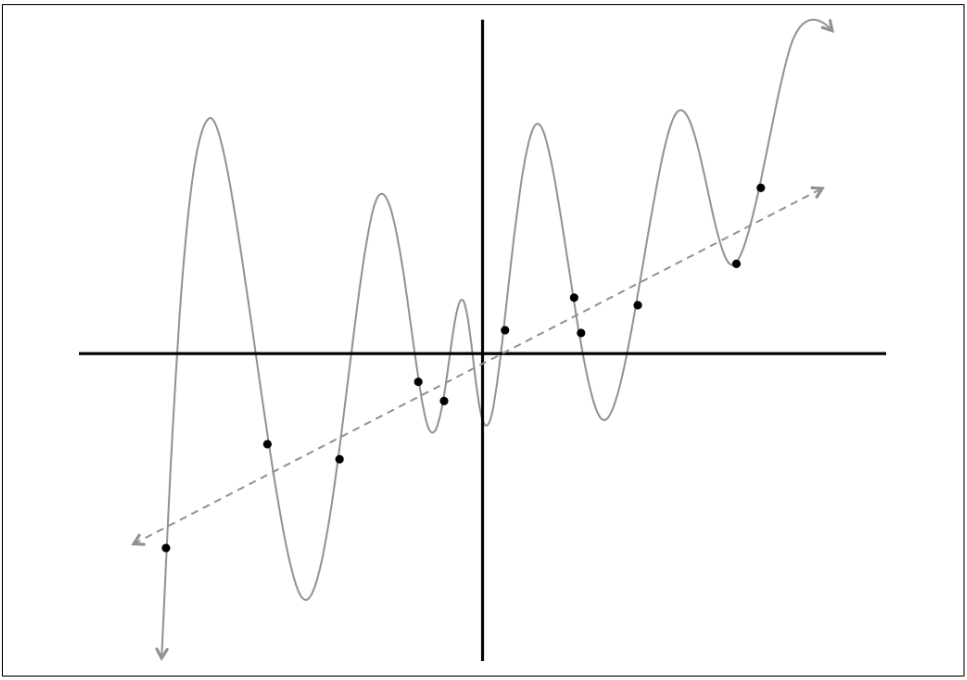


Figure 2-8. Two potential models that might describe our dataset: a linear model versus a degree 12 polynomial

We are given a bunch of data points on a flat plane, and our goal is to find a curve that best describes this dataset (i.e., will allow us to predict the y-coordinate of a new point given its x-coordinate). Using the data, we train two different models: a linear model and a degree 12 polynomial. Which curve should we trust? The line which gets almost no training example correctly? Or the complicated curve that hits every single point in the dataset? At this point we might trust the linear fit because it seems much less contrived. But just to be sure, let's add more data to our dataset! The result is shown in Figure 2-9.

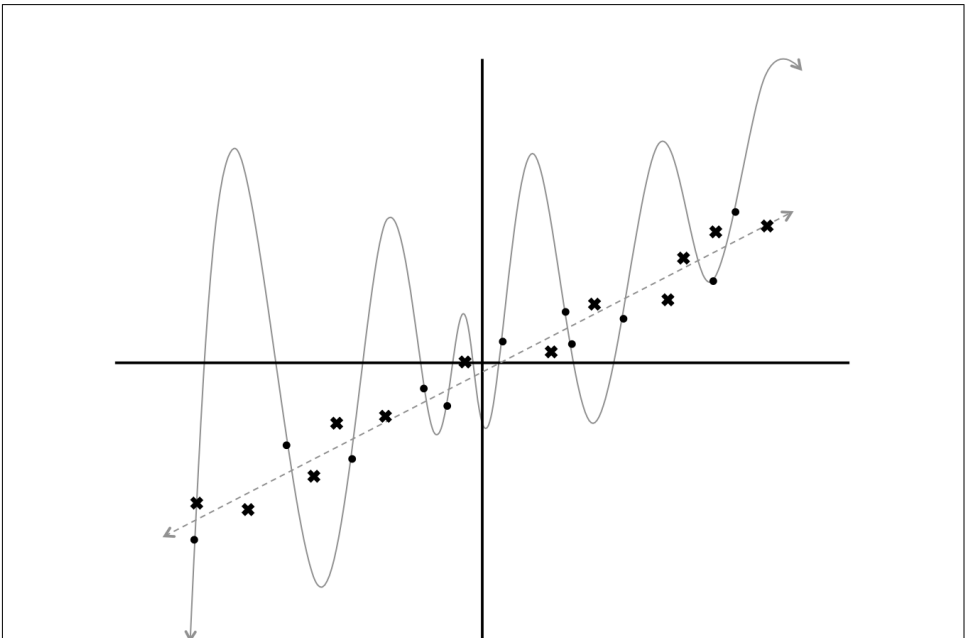


Figure 2-9. Evaluating our model on new data indicates that the linear fit is a much better model than the degree 12 polynomial

Now the verdict is clear: the linear model is not only better subjectively but also quantitatively (measured using the squared error metric). But this leads to a very interesting point about training and evaluating machine learning models. By building a very complex model, it's quite easy to perfectly fit our training dataset because we give our model enough degrees of freedom to contort itself to fit the observations in the training set. But when we evaluate such a complex model on new data, it performs very poorly. In other words, the model does not *generalize* well. This is a phenomenon called *overfitting*, and it is one of the biggest challenges that a machine learning engineer must combat. This becomes an even more significant issue in deep learning, where our neural networks have large numbers of layers containing many neurons. The number of connections in these models is astronomical, reaching the millions. As a result, overfitting is commonplace.

Let's see how this looks in the context of a neural network. Let's say we have a neural network with two inputs, a softmax output of size two, and a hidden layer with 3, 6,

or 20 neurons. We train these networks using mini-batch gradient descent (batch size 10), and the results, visualized using ConvNetJS, are shown in Figure 2-10.[3]

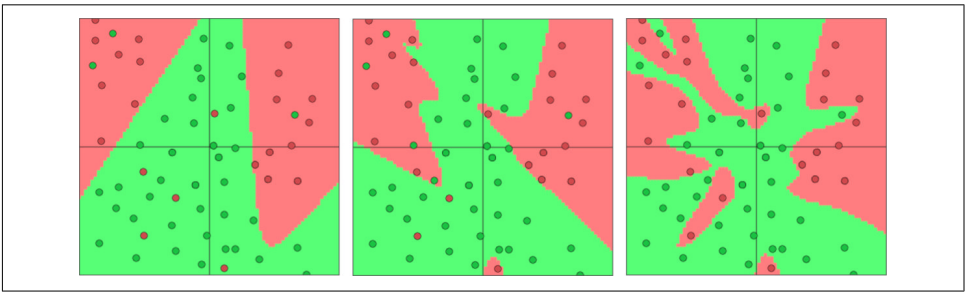


Figure 2-10. A visualization of neural networks with 3, 6, and 20 neurons (in that order) in their hidden layer

It's already quite apparent from these images that as the number of connections in our network increases, so does our propensity to overfit to the data. We can similarly see the phenomenon of overfitting as we make our neural networks deep. These results are shown in Figure 2-11, where we use networks that have one, two, or four hidden layers of three neurons each.

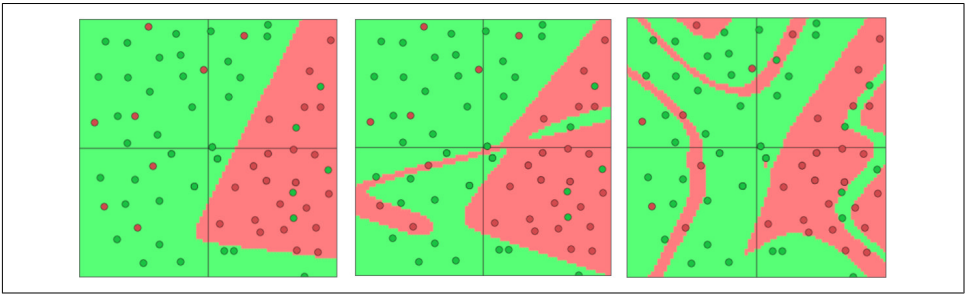


Figure 2-11. A visualization of neural networks with one, two, and four hidden layers (in that order) of three neurons each

This leads to three major observations. First, the machine learning engineer is always working with a direct trade-off between overfitting and model complexity. If the model isn't complex enough, it may not be powerful enough to capture all of the useful information necessary to solve a problem. However, if our model is very complex (especially if we have a limited amount of data at our disposal), we run the risk of overfitting. Deep learning takes the approach of solving very complex problems with complex models and taking additional countermeasures to prevent overfitting. We'll see a lot of these measures in this chapter as well as in later chapters.

3 *http://stanford.io/2pOdNhy*

Second, it is very misleading to evaluate a model using the data we used to train it. Using the example in Figure 2-8, this would falsely suggest that the degree 12 polynomial model is preferable to a linear fit. As a result, we almost never train our model on the entire dataset. Instead, as shown in Figure 2-12, we split up our data into a *training set* and a *test set*.

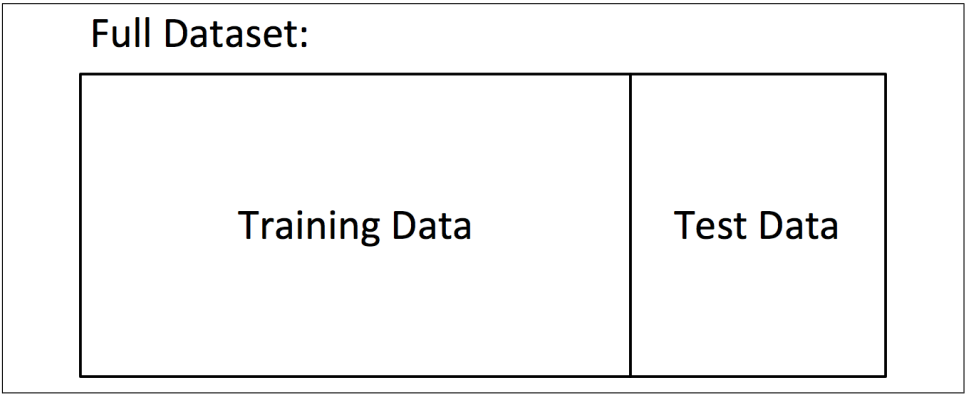


Figure 2-12. We often split our data into nonoverlapping training and test sets in order to fairly evaluate our model

This enables us to make a fair evaluation of our model by directly measuring how well it generalizes on new data it has not yet seen. In the real world, large datasets are hard to come by, so it might seem like a waste to not use all of the data at our disposal during the training process. Consequently, it may be very tempting to reuse training data for testing or cut corners while compiling test data. Be forewarned: if the test set isn't well constructed, we won't be able draw any meaningful conclusions about our model.

Third, it's quite likely that while we're training our data, there's a point in time where instead of learning useful features, we start overfitting to the training set. To avoid that, we want to be able to stop the training process as soon as we start overfitting, to prevent poor generalization. To do this, we divide our training process into *epochs*. An epoch is a single iteration over the entire training set. In other words, if we have a training set of size d and we are doing mini-batch gradient descent with batch size b, then an epoch would be equivalent to $\frac{d}{b}$ model updates. At the end of each epoch, we want to measure how well our model is generalizing. To do this, we use an additional *validation set*, which is shown in Figure 2-13. At the end of an epoch, the validation set will tell us how the model does on data it has yet to see. If the accuracy on the training set continues to increase while the accuracy on the validation set stays the same (or decreases), it's a good sign that it's time to stop training because we're overfitting.

The validation set is also helpful as a proxy measure of accuracy during the process of *hyperparameter optimization*. We've covered several hyperparameters so far in our discussion (learning rate, minibatch size, etc.), but we have yet to develop a framework for how to find the optimal values for these hyperparameters. One potential way to find the optimal setting of hyperparameters is by applying a *grid search,* where we pick a value for each hyperparameter from a finite set of options (e.g., $\epsilon \in \{0.001, 0.01, 0.1\}$, batch size $\in \{16, 64, 128\}$, ...), and train the model with every possible permutation of hyperparameter choices. We elect the combination of hyperparameters with the best performance on the validation set, and report the accuracy of the model trained with best combination on the test set.[4]

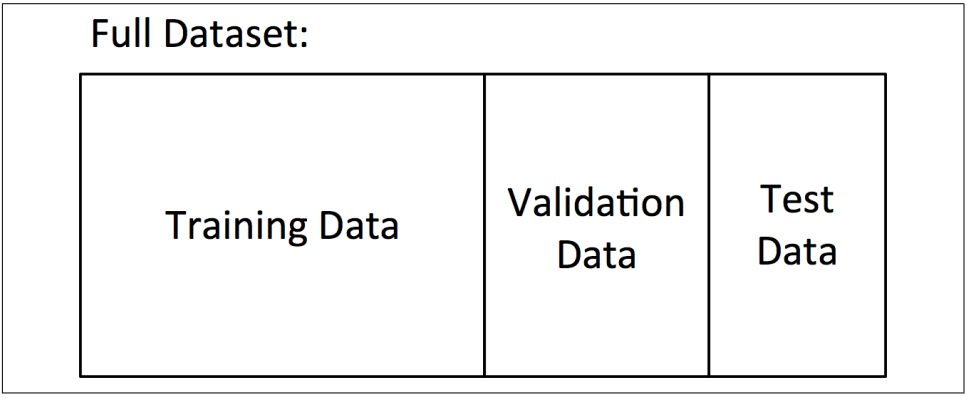


Figure 2-13. In deep learning we often include a validation set to prevent overfitting during the training process

With this in mind, before we jump into describing the various ways to directly combat overfitting, let's outline the workflow we use when building and training deep learning models. The workflow is described in detail in Figure 2-14. It is a tad intricate, but it's critical to understand the pipeline in order to ensure that we're properly training our neural networks.

First we define our problem rigorously. This involves determining our inputs, the potential outputs, and the vectorized representations of both. For instance, let's say our goal was to train a deep learning model to identify cancer. Our input would be an RBG image, which can be represented as a vector of pixel values. Our output would be a probability distribution over three mutually exclusive possibilities: 1) normal, 2) benign tumor (a cancer that has yet to metastasize), or 3) malignant tumor (a cancer that has already metastasized to other organs).

4 Nelder, John A., and Roger Mead. "A simplex method for function minimization." *The Computer Journal* 7.4 (1965): 308-313.

After we define our problem, we need to build a neural network architecture to solve it. Our input layer would have to be of appropriate size to accept the raw data from the image, and our output layer would have to be a softmax of size 3. We will also have to define the internal architecture of the network (number of hidden layers, the connectivities, etc.). We'll further discuss the architecture of image recognition models when we talk about convolutional neural networks in Chapter 4. At this point, we also want to collect a significant amount of data for training or modeling. This data would probably be in the form of uniformly sized pathological images that have been labeled by a medical expert. We shuffle and divide this data up into separate training, validation, and test sets.

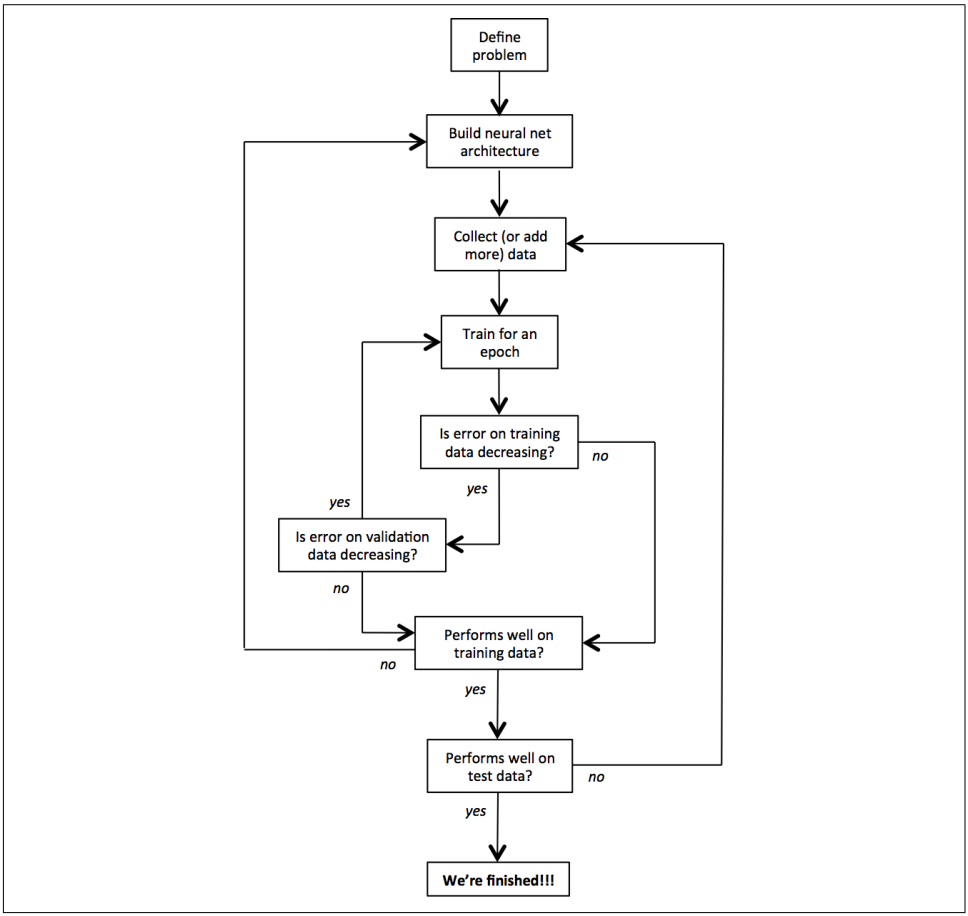


Figure 2-14. Detailed workflow for training and evaluating a deep learning model

Finally, we're ready to begin gradient descent. We train the model on our training set for an epoch at a time. At the end of each epoch, we ensure that our error on the training set and validation set is decreasing. When one of these stops to improve, we

terminate and make sure we're happy with the model's performance on the test data. If we're unsatisfied, we need to rethink our architecture or reconsider whether the data we collect has the information required to make the prediction we're interested in making. If our training set error stopped improving, we probably need to do a better job of capturing the important features in our data. If our validation set error stopped improving, we probably need to take measures to prevent overfitting.

If, however, we are happy with the performance of our model on the training data, then we can measure its performance on the test data, which the model has never seen before this point. If it is unsatisfactory, we need more data in our dataset because the test set seems to consist of example types that weren't well represented in the training set. Otherwise, we are finished!

Preventing Overfitting in Deep Neural Networks

There are several techniques that have been proposed to prevent overfitting during the training process. In this section, we'll discuss these techniques in detail.

One method of combatting overfitting is called *regularization*. Regularization modifies the objective function that we minimize by adding additional terms that penalize large weights. In other words, we change the objective function so that it becomes $Error + \lambda f(\theta)$, where $f(\theta)$ grows larger as the components of θ grow larger, and λ is the regularization strength (another hyperparameter). The value we choose for λ determines how much we want to protect against overfitting. A $\lambda = 0$ implies that we do not take any measures against the possibility of overfitting. If λ is too large, then our model will prioritize keeping θ as small as possible over trying to find the parameter values that perform well on our training set. As a result, choosing λ is a very important task and can require some trial and error.

The most common type of regularization in machine learning is *L2 regularization*.[5] It can be implemented by augmenting the error function with the squared magnitude of all weights in the neural network. In other words, for every weight w in the neural network, we add $\frac{1}{2}\lambda w^2$ to the error function. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. This has the appealing property of encouraging the network to use all of its inputs a little rather than using only some of its inputs a lot. Of particular note is that during the gradient descent update, using the L2 regularization ultimately means that every weight is decayed linearly to zero. Because of this phenomenon, L2 regularization is also commonly referred to as *weight decay*.

5 Tikhonov, Andrei Nikolaevich, and Vladlen Borisovich Glasko. "Use of the regularization method in nonlinear problems." *USSR Computational Mathematics and Mathematical Physics* 5.3 (1965): 93-107.

We can visualize the effects of L2 regularization using ConvNetJS. Similar to Figures 2-10 and 2-11, we use a neural network with 2 inputs, a softmax output of size 2, and a hidden layer with 20 neurons. We train the networks using mini-batch gradient descent (batch size 10) and regularization strengths of 0.01, 0.1, and 1. The results can be seen in Figure 2-15.

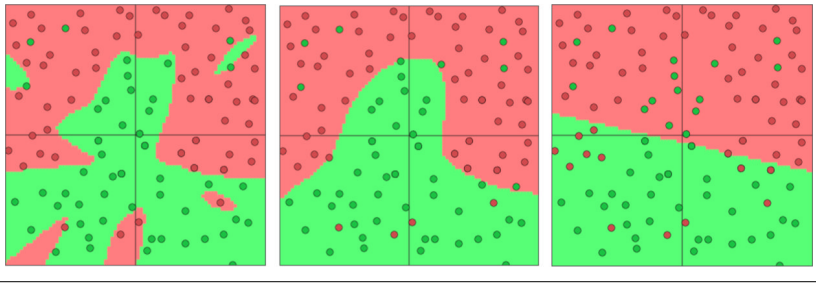


Figure 2-15. A visualization of neural networks trained with regularization strengths of 0.01, 0.1, and 1 (in that order)

Another common type of regularization is *L1 regularization.* Here, we add the term $\lambda|w|$ for every weight w in the neural network. The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e., very close to exactly zero). In other words, neurons with L1 regularization end up using only a small subset of their most important inputs and become quite resistant to noise in the inputs. In comparison, weight vectors from L2 regularization are usually diffuse, small numbers. L1 regularization is very useful when you want to understand exactly which features are contributing to a decision. If this level of feature analysis isn't necessary, we prefer to use L2 regularization because it empirically performs better.

Max norm constraints have a similar goal of attempting to restrict θ from becoming too large, but they do this more directly.[6] Max norm constraints enforce an absolute upper bound on the magnitude of the incoming weight vector for every neuron and use projected gradient descent to enforce the constraint. In other words, any time a gradient descent step moves the incoming weight vector such that $||w||_2 > c$, we project the vector back onto the ball (centered at the origin) with radius c. Typical values of c are 3 and 4. One of the nice properties is that the parameter vector cannot grow out of control (even if the learning rates are too high) because the updates to the weights are always bounded.

6 Srebro, Nathan, Jason DM Rennie, and Tommi S. Jaakkola. "Maximum-Margin Matrix Factorization." *NIPS*, Vol. 17, 2004.

Dropout is a very different kind of method for preventing overfitting that has become one of the most favored methods of preventing overfitting in deep neural networks. [7] While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. Intuitively, this forces the network to be accurate even in the absence of certain information. It prevents the network from becoming too dependent on any one (or any small combination) of neurons. Expressed more mathematically, it prevents overfitting by providing a way of approximately combining exponentially many different neural network architectures efficiently. The process of dropout is expressed pictorially in Figure 2-16.

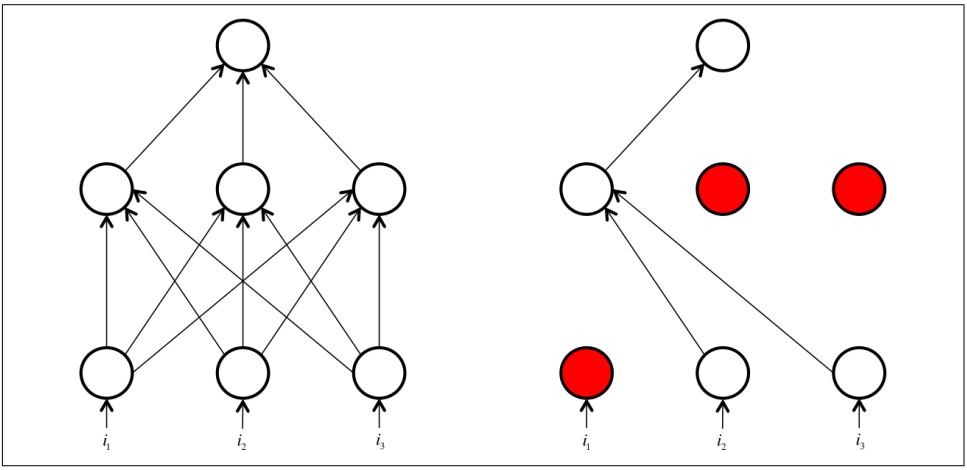


Figure 2-16. Dropout sets each neuron in the network as inactive with some random probability during each minibatch of training

Dropout is pretty intuitive to understand, but there are some important intricacies to consider. First, we'd like the outputs of neurons during test time to be equivalent to their expected outputs at training time. We could fix this naïvely by scaling the output at test time. For example, if $p = 0.5$, neurons must halve their outputs at test time in order to have the same (expected) output they would have during training. This is easy to see because a neuron's output is set to 0 with probability $1 - p$. This means that if a neuron's output prior to dropout was x, then after dropout, the expected output would be $E[\text{output}] = px + (1 - p) \cdot 0 = px$. This naïve implementation of dropout is undesirable, however, because it requires scaling of neuron outputs at test time. Test-time performance is extremely critical to model evaluation, so it's always preferable to use *inverted dropout*, where the scaling occurs at training time instead of at test

7 Srivastava, Nitish, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

time. In inverted dropout, any neuron whose activation hasn't been silenced has its output divided by p before the value is propagated to the next layer. With this fix, $E[\text{output}] = p \cdot \frac{x}{p} + (1 - p) \cdot 0 = x$, and we can avoid arbitrarily scaling neuronal output at test time.

Summary

In this chapter, we've learned all of the basics involved in training feed-forward neural networks. We've talked about gradient descent, the backpropagation algorithm, as well as various methods we can use to prevent overfitting. In the next chapter, we'll put these lessons into practice when we use the TensorFlow library to efficiently implement our first neural networks. Then in Chapter 4, we'll return to the problem of optimizing objective functions for training neural networks and design algorithms to significantly improve performance. These improvements will enable us to process much more data, which means we'll be able to build more comprehensive models.

CHAPTER 3

Implementing Neural Networks in TensorFlow

What Is TensorFlow?

Although we could spend this entire book describing deep learning models in the abstract, we hope that by the end of this text, you not only have an understanding of how deep models work, but also that you are equipped with the skill set required to build these models from scratch for your own problem spaces. Now that we have a better theoretical understanding of deep learning models, we will spend this chapter implementing some of these algorithms in software.

The primary tool that we will use throughout this text is called TensorFlow.[1] TensorFlow is an open source software library released in 2015 by Google to make it easier for developers to design, build, and train deep learning models. TensorFlow originated as an internal library that Google developers used to build models in-house, and we expect additional functionality to be added to the open source version as it is tested and vetted in the internal flavor. Although TensorFlow is only one of several options available to developers, we choose to use it here because of its thoughtful design and ease of use. We'll briefly compare TensorFlow to alternatives in the next section.

On a high level, TensorFlow is a Python library that allows users to express arbitrary computation as a graph of *data flows*. Nodes in this graph represent mathematical operations, whereas edges represent data that is communicated from one node to another. Data in TensorFlow is represented as tensors, which are multidimensional arrays (representing vectors with a 1D tensor, matrices with a 2D tensor, etc.).

1 *https://www.tensorflow.org/*

Although this framework for thinking about computation is valuable in many different fields, TensorFlow is primarily used for deep learning in practice and research.

Thinking about neural networks as tensors and vice versa isn't trivial, but rather a skill that we will develop through the course of this text. Representing deep neural networks in this way allows us to take advantage of the speedups afforded by modern hardware (i.e., GPU acceleration of parallel tensor operations) and provides us with a clean, but expressive, method for implementing models. In this chapter, we will discuss the basics of TensorFlow and walk through two simple examples (logistic regression and multilayer feed-forward neural networks). But before we dive in, let's talk a little bit about how TensorFlow stacks up against other frameworks for representing deep learning models.

How Does TensorFlow Compare to Alternatives?

In addition to TensorFlow, there are a number of libraries that have popped up over the years for building deep neural networks. These include Theano, Torch, Caffe, Neon, and Keras.[2] Based on two simple criteria (expressiveness and presence of an active developer community), we ultimately narrowed the field of options to TensorFlow, Theano (built by the LISA Lab out of the University of Montreal), and Torch (largely maintained by Facebook AI Research).

All three of these options boast a hefty developer community, enable users to manipulate tensors with few restrictions, and feature automatic differentiation (which enables users to train deep models without having to crank out the backpropagation algorithms for arbitrary architectures, as we had to do in the previous chapter). One of the drawbacks of Torch, however, is that the framework is written in Lua. Lua is a scripting language much like Python, but is less commonly used outside the deep learning community. We wanted to avoid forcing newcomers to learn a whole new language to build deep learning models, so we further narrowed our options to TensorFlow and Theano.

Between these two options, the decision was difficult (and in fact, an early version of this chapter was first written using Theano), but we chose TensorFlow in the end for several subtle reasons. First, Theano has an additional "graph compilation" step that took significant amounts of time while setting up certain kinds of deep learning architectures. While small in comparison to train time, this compilation phase proved frustrating while writing and debugging new code. Second, TensorFlow has a much cleaner interface as compared to Theano. Many classes of models can be expressed in significantly fewer lines without sacrificing the expressiveness of the framework.

2 *http://deeplearning.net/software/theano/*; *http://torch.ch/*; *http://caffe.berkeleyvision.org/*; *https://www.nervanasys.com/technology/neon/*; *https://keras.io/*

Finally, TensorFlow was built with production use in mind, whereas Theano was designed by researchers almost purely for research purposes. As a result, TensorFlow has many features out of the box and in the works that make it a better choice for real systems (the ability to run in mobile environments, easily build models that span multiple GPUs on a single machine, and train large-scale networks in a distributed fashion). Although familiarity with Theano and Torch can be extremely helpful while navigating open source examples, overviews of these frameworks are beyond the scope of this book.

Installing TensorFlow

Installing TensorFlow in your local development environment is straightforward if you aren't planning on modifying the TensorFlow source code. We use a Python package installation manager called Pip. If you don't already have Pip installed on your computer, use the following commands in your terminal:

```
# Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev

# Mac OS X
$ sudo easy_install pip
```

Once we have Pip (version 8.1 or later) installed on our computers, we can use the following commands to install TensorFlow. Note the difference in Pip package naming if we would like to install a GPU-enabled version of TensorFlow (which we strongly recommend):

```
$ pip install --upgrade tensorflow      # for Python 2.7
$ pip3 install --upgrade tensorflow     # for Python 3.n
$ pip install --upgrade tensorflow-gpu  # for Python 2.7
                                        # and GPU
$ pip3 install --upgrade tensorflow-gpu # for Python 3.n
                                        # and GPU
```

If you installed the GPU-enabled version of TensorFlow, you'll also have to take a couple of additional steps. Specifically, you'll have to download the CUDA Toolkit 8.0[3] and the latest CUDNN Toolkit.[4] Install the CUDA Toolkit 7.0 into */usr/local/cuda*. Then uncompress and copy the CUDNN files into the toolkit directory. Assuming the toolkit is installed in*/usr/local/cuda*, you can follow these instructions to accomplish this:

```
$ tar xvzf cudnn-version-os.tgz
$ sudo cp cudnn-version-os/cudnn.h /usr/local/cuda/include
$ sudo cp cudnn-version-os/libcudnn* /usr/local/cuda/lib64
```

3 *http://docs.nvidia.com/cuda*

4 *https://developer.nvidia.com/rdp/cudnn-archive*

You will also need to set the `LD_LIBRARY_PATH` and `CUDA_HOME` environment variables to give TensorFlow access to your CUDA installation. Consider adding the commands below to your *~/.bash_profile*. These assume your CUDA installation is in */usr/local/cuda*:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64"
export CUDA_HOME=/usr/local/cuda
```

Note that to see these changes appropriately reflected in your current terminal session, you'll have to run:

```
$ source ~/.bash_profile
```

You should now be able to run TensorFlow from your Python shell of choice. In this tutorial, we choose to use IPython. Using Pip, installing IPython only requires the following command:

```
$ pip install ipython
```

Then we can test that our installation of TensorFlow functions as expected:

```
$ ipython

...

In [1]: import tensorflow as tf

In [2]: deep_learning = tf.constant('Deep Learning')

In [3]: session = tf.Session()

In [4]: session.run(deep_learning)
Out[4]: 'Deep Learning'

In [5]: a = tf.constant(2)

In [6]: a = tf.constant(2)

In [7]: multiply = tf.mul(a, b)

In [7]: session.run(multiply)
Out[7]: 6
```

Additional, up-to-date instructions and details about installation can be found on the TensorFlow website.[5]

5 *https://www.tensorflow.org/install/*

Creating and Manipulating TensorFlow Variables

When we build a deep learning model in TensorFlow, we use variables to represent the parameters of the model. TensorFlow variables are in-memory buffers that contain tensors; but unlike normal tensors that are only instantiated when a graph is run and that are immediately wiped clean afterward, variables survive across multiple executions of a graph. As a result, TensorFlow variables have the following three properties:

- Variables must be explicitly initialized before a graph is used for the first time.
- We can use gradient methods to modify variables after each iteration as we search for a model's optimal parameter settings.
- We can save the values stored in variables to disk and restore them for later use.

These three properties are what make TensorFlow especially useful for building machine learning models.

Creating a variable is simple, and TensorFlow provides mechanics that allow us to initialize variables in several ways. Let's start off by initializing a variable that describes the weights connecting neurons between two layers of a feed-forward neural network:

```
weights = tf.Variable(tf.random_normal([300, 200], stddev=0.5),
                      name="weights")
```

Here we pass two arguments to `tf.Variable`.[6] The first, `tf.random_normal`,[7] is an operation that produces a tensor initialized using a normal distribution with standard deviation 0.5. We've specified that this tensor is of size 300 x 200, implying that the weights connect a layer with 300 neurons to a layer with 200 neurons. We've also passed a name to our call to `tf.Variable`. The name is a unique identifier that allows us to refer to the appropriate node in the computation graph. In this case, weights is meant to be *trainable*; or in other words, we will automatically compute and apply gradients to `weights`. If `weights` is not meant to be trainable, we may pass an optional flag when we call `tf.Variable`:

```
weights = tf.Variable(tf.random_normal([300, 200], stddev=0.5),
                      name="weights", trainable=False)
```

In addition to using `tf.random_normal`, there are several other methods to initialize a TensorFlow variable:

---

6 *https://www.tensorflow.org/api_docs/python/tf/Variable*

7 *https://www.tensorflow.org/api_docs/python/tf/random_normal*

```
# Common tensors from the TensorFlow API docs

tf.zeros(shape, dtype=tf.float32, name=None)
tf.ones(shape, dtype=tf.float32, name=None)
tf.random_normal(shape, mean=0.0, stddev=1.0,
                 dtype=tf.float32, seed=None,
                 name=None)
tf.truncated_normal(shape, mean=0.0, stddev=1.0,
                    dtype=tf.float32, seed=None,
                    name=None)
tf.random_uniform(shape, minval=0, maxval=None,
                  dtype=tf.float32, seed=None,
                  name=None)
```

When we call `tf.Variable`, three operations are added to the computation graph:

- The operation producing the tensor we use to initialize our variable
- The `tf.assign` operation, which is responsible for filling the variable with the initializing tensor prior to the variable's use
- The variable operation, which holds the current value of the variable

This can be visualized as shown in Figure 3-1.

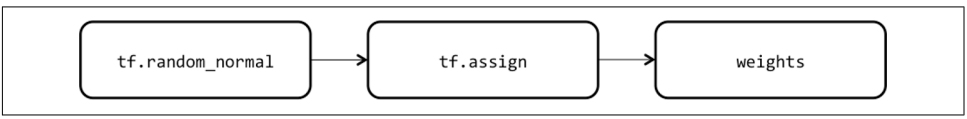


Figure 3-1. Three operations are added to the graph when instantiating a TensorFlow variable. In this example, we instantiate the variable weights using a random normal initializer.

As we mentioned previously in the three operations, before we use any TensorFlow variable, the `tf.assign`[8] operation must be run so that the variable is appropriately initialized with the desired value. We can do this by running `tf.initial ize_all_variables()`,[9] which will trigger all of the `tf.assign` operations in our graph. We can also selectively initialize only certain variables in our computational graph using the `tf.initialize_variables(var1, var2, ...)`.[10] We'll describe this in more detail when we discuss sessions in TensorFlow.

8 *https://www.tensorflow.org/api_docs/python/tf/assign*

9 *http://bit.ly/2rtqoIA*

10 *https://www.tensorflow.org/api_docs/python/tf/initialize_variables*

TensorFlow Operations

We've already talked a little bit about operations in the context of variable initialization, but these only make up a small subset of the universe of operations available in TensorFlow. On a high level, TensorFlow *operations* represent abstract transformations that are applied to tensors in the computation graph. Operations may have attributes that may be supplied a priori or are inferred at runtime. For example, an attribute may serve to describe the expected types of input (adding tensors of type `float32` versus `int32`). Just as variables are named, operations may also be supplied with an optional name attribute for easy reference into the computation graph.

An operation consists of one or more *kernels*, which represent device-specific implementations. For example, an operation may have separate CPU and GPU kernels because it can be more efficiently expressed on a GPU. This is the case for many TensorFlow operations on matrices.

To provide an overview of the types of operations available, we include Table 3-1 from the original TensorFlow white paper detailing the various categories of operations in TensorFlow.[11]

Table 3-1. A summary table of TensorFlow operations

Category	Examples
Element-wise mathematical operations	`Add`, `Sub`, `Mul`, `Div`, `Exp`, `Log`, `Greater`, `Less`, `Equal`, ...
Array operations	`Concat`, `Slice`, `Split`, `Constant`, `Rank`, `Shape`, `Shuffle`, ...
Matrix operations	`MatMul`, `MatrixInverse`, `MatrixDeterminant`, ...
Stateful operations	`Variable`, `Assign`, `AssignAdd`, ...
Neural network building blocks	`SoftMax`, `Sigmoid`, `ReLU`, `Convolution2D`, `MaxPool`, ...
Checkpointing operations	`Save`, `Restore`
Queue and synchronization operations	`Enqueue`, `Dequeue`, `MutexAcquire`, `MutexRelease`, ...
Control flow operations	`Merge`, `Switch`, `Enter`, `Leave`, `NextIteration`

Placeholder Tensors

Now that we have a solid understanding of TensorFlow variables and operations, we have a nearly complete description of the components of a TensorFlow computation graph. The only missing piece is how we pass the input to our deep model (during both train and test time). A variable is insufficient because it is only meant to be initialized once. Instead, we need a component that we populate every single time the computation graph is run.

11 Abadi, Martín, et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." *arXiv preprint arXiv*:1603.04467 (2016).

TensorFlow solves this problem using a construct called a *placeholder*.[12] A placeholder is instantiated as follows and can be used in operations just like ordinary TensorFlow variables and tensors:

```
x = tf.placeholder(tf.float32, name="x", shape=[None, 784])
W = tf.Variable(tf.random_uniform([784,10], -1, 1), name="W")
multiply = tf.matmul(x, W)
```

Here we define a placeholder where `x` represents a minibatch of data stored as `float32`'s. We notice that `x` has 784 columns, which means that each data sample has 784 dimensions. We also notice that `x` has an undefined number of rows. This means that x can be initialized with an arbitrary number of data samples. While we could instead multiply each data sample separately by `W`, expressing a full minibatch as a tensor allows us to compute the results for all the data samples in parallel. The result is that the i^{th} row of the `multiply` tensor corresponds to `W` multiplied with the i^{th} data sample.

Just as variables need to be initialized the first time the computation graph is built, placeholders need to be filled every time the computation graph (or a subgraph) is run. We'll discuss how this works in more detail in the next section.

Sessions in TensorFlow

A TensorFlow program interacts with a computation graph using a *session*.[13] The TensorFlow session is responsible for building the initial graph, and can be used to initialize all variables appropriately and to run the computational graph. To explore each of these pieces, let's consider the following simple Python script:

```
import tensorflow as tf
from read_data import get_minibatch()

x = tf.placeholder(tf.float32, name="x", shape=[None, 784])
W = tf.Variable(tf.random_uniform([784, 10], -1, 1), name="W")
b = tf.Variable(tf.zeros([10]), name="biases")
output = tf.matmul(x, W) + b

init_op = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init_op)
feed_dict = {"x" : get_minibatch()}
sess.run(output, feed_dict=feed_dict)
```

12 *https://www.tensorflow.org/api_docs/python/tf/placeholder*

13 *https://www.tensorflow.org/api_docs/python/tf/Session*

The first four lines after the `import` statement describe the computational graph that is built by the session when it is finally instantiated. The graph (sans variable initialization operations) is depicted in Figure 3-2. We then initialize the variables as required by using the session variable to run the initialization operation in `sess.run(init_op)`. Finally, we can run the subgraph by calling `sess.run` again, but this time we pass in the tensors (or list of tensors) we want to compute along with a `feed_dict` that fills the placeholders with the necessary input data.

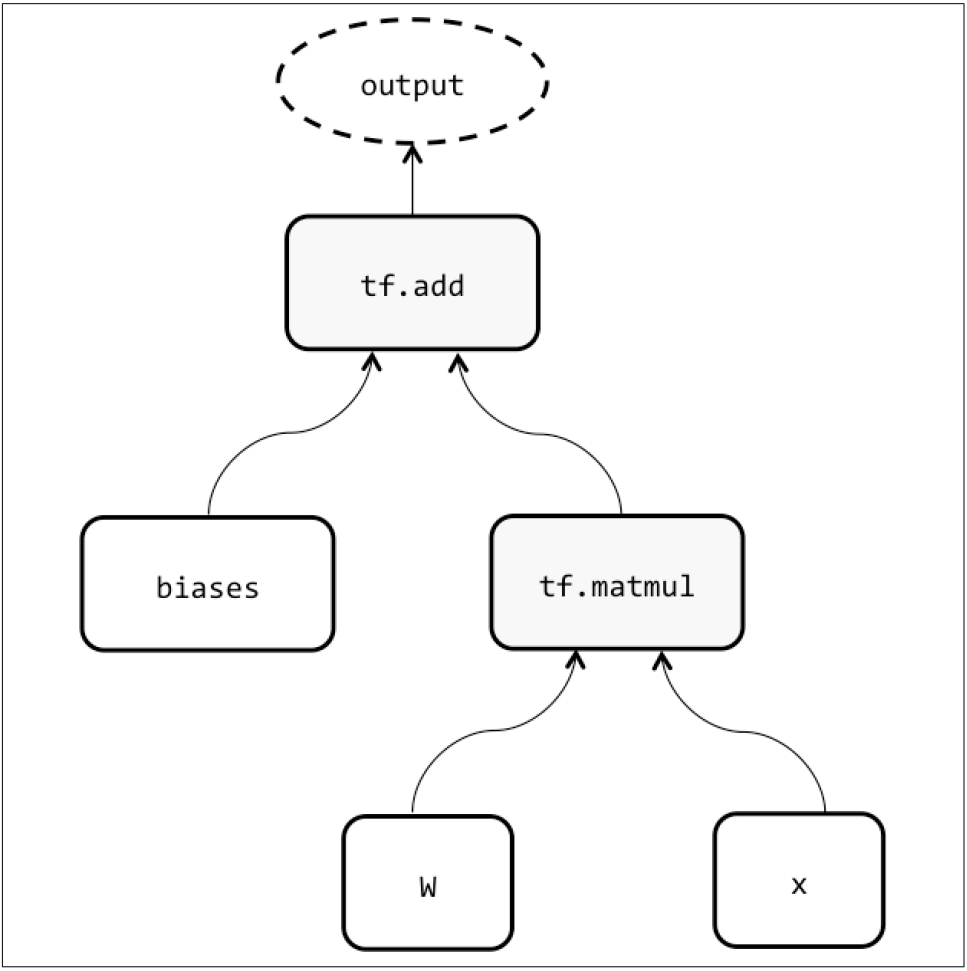


Figure 3-2. This is a an example of a simple computational graph in TensorFlow

Finally, the `sess.run` interface can also be used to train networks. We will explore this in further detail when we use TensorFlow to train our first machine learning model on MNIST. But how exactly does a single line of code (`sess.run`) accomplish such a wide variety of functions? The answer lies in the powerful expressivity of the underly-

ing computational graph. All of these functionalities are represented as TensorFlow operations that can be passed as arguments to `sess.run`. All `sess.run` needs to do is traverse down the computational graph to identify all of the dependencies that compose the relevant subgraph, ensure that all of the placeholder variables that belong to the identified subgraph are filled using the `feed_dict`, and then traverse back up the subgraph (executing all of the intermediate operations) to evaluate the original arguments.

Now that we have a comprehensive understanding of sessions and how to run them, we'll explore two more major concepts in building and maintaining computational graphs.

Navigating Variable Scopes and Sharing Variables

Although we won't run into this problem just yet, building complex models often requires reusing and sharing large sets of variables that we'll want to instantiate together in one place. Unfortunately, trying to enforce modularity and readability can result in unintended results if we aren't careful. Let's consider the following example:

```
def my_network(input):
     W_1 = tf.Variable(tf.random_uniform([784, 100], -1, 1),
                       name="W_1")
     b_1 = tf.Variable(tf.zeros([100]), name="biases_1")
     output_1 = tf.matmul(input, W_1) + b_1

     W_2 = tf.Variable(tf.random_uniform([100, 50], -1, 1),
                       name="W_2")
     b_2 = tf.Variable(tf.zeros([50]), name="biases_2")
     output_2 = tf.matmul(output_1, W_2) + b_2

     W_3 = tf.Variable(tf.random_uniform([50, 10], -1, 1),
                       name="W_3")
     b_3 = tf.Variable(tf.zeros([10]), name="biases_3")
     output_3 = tf.matmul(output_2, W_3) + b_3

     # printing names
     print "Printing names of weight parameters"
     print W_1.name, W_2.name, W_3.name
     print "Printing names of bias parameters"
     print b_1.name, b_2.name, b_3.name

     return output_3
```

This network setup consists of six variables describing three layers. As a result, if we wanted to use this network multiple times, we'd prefer to encapsulate it into a compact function like `my_network`, which we can call multiple times. However, when we try to use this network on two different inputs, we get something unexpected:

```
In [1]: i_1 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_1")

In [2]: my_network(i_1)
Printing names of weight parameters
W_1:0 W_2:0 W_3:0
Printing names of bias parameters
biases_1:0 biases_2:0 biases_3:0
Out[2]: <tensorflow.python.framework.ops.Tensor ...>

In [1]: i_2 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_2")

In [2]: my_network(i_2)
Printing names of weight parameters
W_1_1:0 W_2_1:0 W_3_1:0
Printing names of bias parameters
biases_1_1:0 biases_2_1:0 biases_3_1:0
Out[2]: <tensorflow.python.framework.ops.Tensor ...>
```

If we observe closely, our second call to `my_network` doesn't use the same variables as the first call (in fact, the names are different!). Instead, we've created a second set of variables! In many cases, we don't want to create a copy, but rather reuse the model and its variables. It turns out, that in this case, we shouldn't be using `tf.Variable`. Instead, we should be using a more advanced naming scheme that takes advantage of TensorFlow's variable scoping.

TensorFlow's variable scoping mechanisms are largely controlled by two functions:

`tf.get_variable(<name>, <shape>, <initializer>)`
: Checks if a variable with this name exists, retrieves the variable if it does, or creates it using the shape and initializer if it doesn't.[14]

`tf.variable_scope(<scope_name>)`
: Manages the namespace and determines the scope in which `tf.get_variable` operates.[15]

Let's try to rewrite `my_network` in a cleaner fashion using TensorFlow variable scoping. The new names of our variables are namespaced as `"layer1/W"`, `"layer2/b"`, `"layer2/W"`, and so forth:

14 *https://www.tensorflow.org/api_docs/python/tf/get_variable*

15 *https://www.tensorflow.org/api_docs/python/tf/variable_scope*

```
def layer(input, weight_shape, bias_shape):
     weight_init = tf.random_uniform_initializer(minval=-1,
       maxval=1)
     bias_init = tf.constant_initializer(value=0)
     W = tf.get_variable("W", weight_shape,
                         initializer=weight_init)
     b = tf.get_variable("b", bias_shape,
                         initializer=bias_init)
     return tf.matmul(input, W) + b

def my_network(input):
     with tf.variable_scope("layer_1"):
          output_1 = layer(input, [784, 100], [100])

     with tf.variable_scope("layer_2"):
          output_2 = layer(output_1, [100, 50], [50])

     with tf.variable_scope("layer_3"):
          output_3 = layer(output_2, [50, 10], [10])

     return output_3
```

Now let's try to call `my_network` `twice`, just like we did in the preceding code block:

```
In [1]: i_1 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_1")

In [2]: my_network(i_1)
Out[2]: <tensorflow.python.framework.ops.Tensor ...>

In [1]: i_2 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_2")

In [2]: my_network(i_2)
ValueError: Over-sharing: Variable layer_1/W already exists...
```

Unlike `tf.Variable`, the `tf.get_variable` command checks that a variable of the given name hasn't already been instantiated. By default, sharing is not allowed (just to be safe!), but if we want to enable sharing within a variable scope, we can say so explicitly:

```
with tf.variable_scope("shared_variables") as scope:
     i_1 = tf.placeholder(tf.float32, [1000, 784], name="i_1")
     my_network(i_1)
     scope.reuse_variables()
     i_2 = tf.placeholder(tf.float32, [1000, 784], name="i_2")
     my_network(i_2)
```

This allows us to retain modularity while still allowing variable sharing. And as a nice byproduct, our naming scheme is cleaner as well.

Managing Models over the CPU and GPU

TensorFlow allows us to utilize multiple computing devices, if we so desire, to build and train our models. Supported devices are represented by string IDs and normally consist of the following:

`"/cpu:0"`
: The CPU of our machine.

`"/gpu:0"`
: The first GPU of our machine, if it has one.

`"/gpu:1"`
: The second GPU of our machine, if it has one.

When a TensorFlow operation has both CPU and GPU kernels, and GPU use is enabled, TensorFlow will automatically opt to use the GPU implementation. To inspect which devices are used by the computational graph, we can initialize our TensorFlow session with the `log_device_placement` set to `True`:

```
sess = tf.Session(config=tf.ConfigProto(
                    log_device_placement=True))
```

If we desire to use a specific device, we may do so by using `with tf.device`[16] to select the appropriate device. If the chosen device is not available, however, an error will be thrown. If we would like TensorFlow to find another available device if the chosen device does not exist, we can pass the `allow_soft_placement` flag to the session variable as follows:[17]

```
with tf.device('/gpu:2'):
  a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2], name='a')
  b = tf.constant([1.0, 2.0], shape=[2, 1], name='b')
  c = tf.matmul(a, b)

sess = tf.Session(config=tf.ConfigProto(
      allow_soft_placement=True, log_device_placement=True))

sess.run(c)
```

TensorFlow also allows us to build models that span multiple GPUs by building models in a tower-like fashion as shown in Figure 3-3. The following code is an example of multi-GPU code:

16 *https://www.tensorflow.org/api_docs/python/tf/device*

17 *https://www.tensorflow.org/api_docs/python/tf/ConfigProto*

```
c = []

for d in ['/gpu:0', '/gpu:1']:
  with tf.device(d):
    a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2],
                    name='a')
    b = tf.constant([1.0, 2.0], shape=[2, 1], name='b')
    c.append(tf.matmul(a, b))

with tf.device('/cpu:0'):
  sum = tf.add_n(c)

sess = tf.Session(config=tf.ConfigProto(
                  log_device_placement=True))

sess.run(sum)
```

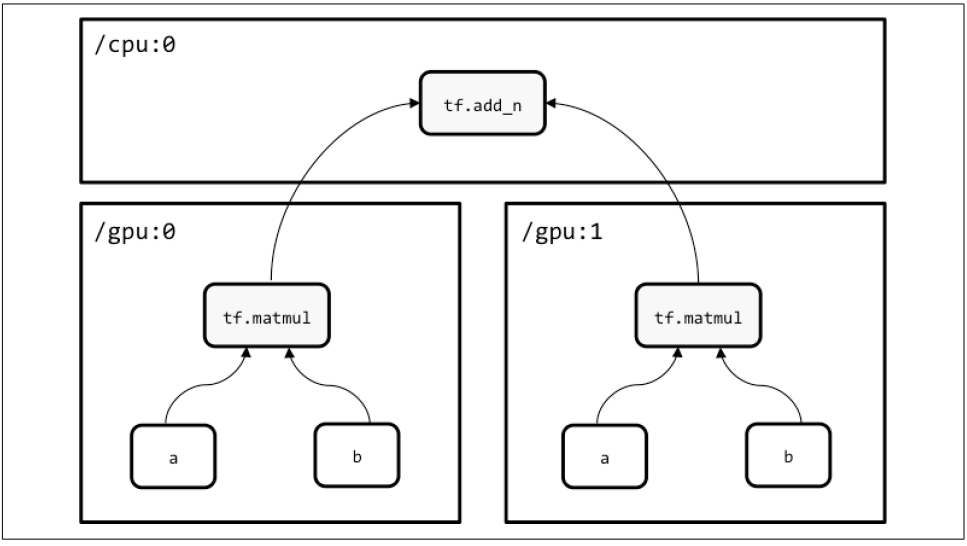


Figure 3-3. Building multi-GPU models in a tower-like fashion

Specifying the Logistic Regression Model in TensorFlow

Now that we've developed all of the basic concepts of TensorFlow, let's build a simple model to tackle the MNIST dataset. As you may recall, our goal is to identify handwritten digits from 28 x 28 black-and-white images. The first network that we'll build implements a simple machine learning algorithm known as logistic regression.[18]

18 Cox, David R. "The Regression Analysis of Binary Sequences." *Journal of the Royal Statistical Society. Series B (Methodological)* (1958): 215-242.

On a high level, logistic regression is a method by which we can calculate the probability that an input belongs to one of the target classes. In our case, we'll compute the probability that a given input image is a 0, 1, ..., or 9. Our model uses a matrix W representing the weights of the connections in the network, as well as a vector b corresponding to the biases to estimate whether an input x belongs to class i using the softmax expression we talked about earlier:

$$P(y = i \mid x) = softmax_i(Wx + b) = \frac{e^{W_i x + b_i}}{\Sigma_j e^{W_j x + b_j}}$$

Our goal is to learn the values for W and b that most effectively classify our inputs as accurately as possible. Pictorially, we can express the logistic regression network as shown in Figure 3-4 (bias connections are not shown to reduce clutter).

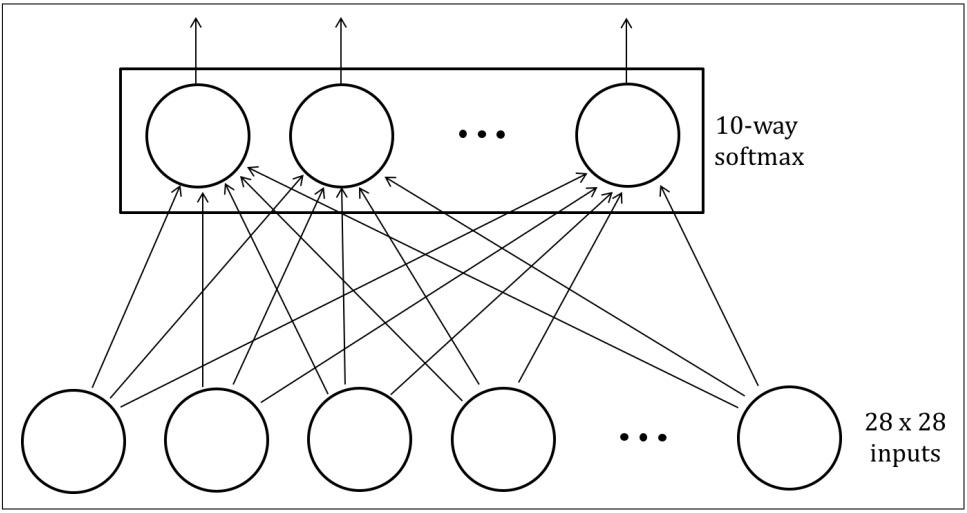


Figure 3-4. Interpreting logistic regression as a primitive neural network

You'll notice that the network interpretation for logistic regression is rather primitive. It doesn't have any hidden layers, meaning that it is limited in its ability to learn complex relationships! We have an output softmax of size 10 because we have 10 possible outcomes for each input. Moreover, we have an input layer of size 784, one input neuron for every pixel in the image! As we'll see, the model makes decent headway toward correctly classifying our dataset, but there's lots of room for improvement. Over the course of the rest of this chapter and Chapter 5, we'll try to significantly improve our accuracy. But first, let's look at how we can implement the logistic network in TensorFlow so we can train it on our computer!

We'll build the the logistic regression model in four phases:

1. `inference`: produces a probability distribution over the output classes given a minibatch
2. `loss`: computes the value of the error function (in this case, the cross-entropy loss)
3. `training`: responsible for computing the gradients of the model's parameters and updating the model
4. `evaluate`: will determine the effectiveness of a model

Given a minibatch, which consists of 784-dimensional vectors representing MNIST images, we can represent logistic regression by taking the softmax of the input multiplied with a matrix representing the weights connecting the input and output layer. Each row of the output tensor represents the probability distribution over output classes for each corresponding data sample in the minibatch:

```
def inference(x):
     tf.constant_initializer(value=0)
     W = tf.get_variable("W", [784, 10],
                         initializer=init)
     b = tf.get_variable("b", [10],
                         initializer=init)
     output = tf.nn.softmax(tf.matmul(x, W) + b)
     return output
```

Now, given the correct labels for a minibatch, we should be able to compute the average error per data sample. We accomplish this using the following code snippet that computes the cross-entropy loss over a minibatch:

```
def loss(output, y):
     dot_product = y * tf.log(output)

    # Reduction along axis 0 collapses each column into a
    # single value, whereas reduction along axis 1 collapses
    # each row into a single value. In general, reduction along
    # axis i collapses the ith dimension of a tensor to size 1.
    xentropy = -tf.reduce_sum(dot_product, reduction_indices=1)

     loss = tf.reduce_mean(xentropy)

     return loss
```

Then, given the current cost incurred, we'll want to compute the gradients and modify the parameters of the model appropriately. TensorFlow makes this easy by giving us access to built-in optimizers that produce a special train operation that we can run via a TensorFlow session when we minimize them. Note that when we create the training operation, we also pass in a variable that represents the number of minibatches that have been processed. Each time the training operation is run, this step variable is incremented so that we can keep track of progress:

```
def training(cost, global_step):
    optimizer = tf.train.GradientDescentOptimizer(
   learning_rate)
    train_op = optimizer.minimize(cost,
   global_step=global_step)
    return train_op
```

Finally, we put together a simple computational subgraph to evaluate the model on the validation or test set:

```
def evaluate(output, y):
    correct_prediction = tf.equal(tf.argmax(output, 1),
                                  tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction,
                                      tf.float32))
    return accuracy
```

This completes TensorFlow graph setup for the logistic regression model.

Logging and Training the Logistic Regression Model

Now that we have all of the major pieces, we begin to stitch them together. In order to log important information as we train the model, we log several summary statistics. For example, we use the `tf.scalar_summary`[19] and `tf.histogram_summary`[20] commands to log the cost for each minibatch, validation error, and the distribution of parameters. For reference, we'll demonstrate the scalar summary statistic for the cost function:

```
def training(cost, global_step):
    tf.scalar_summary("cost", cost)
    optimizer = tf.train.GradientDescentOptimizer(
   learning_rate)
    train_op = optimizer.minimize(cost,
   global_step=global_step)
    return train_op
```

Every epoch, we run the `tf.merge_all_summaries`[21] in order to collect all summary statistics we've logged and use a `tf.train.SummaryWriter` to write the log to disk. In the next section, we'll describe how we can use visualize these logs with the built-in TensorBoard tool.

19 *https://www.tensorflow.org/api_docs/python/tf/summary/scalar*

20 *https://www.tensorflow.org/api_docs/python/tf/summary/histogram*

21 *https://www.tensorflow.org/api_docs/python/tf/summary/merge_all*

In addition to saving summary statistics, we also save the model parameters using the `tf.train.Saver` model saver. By default, the saver maintains the latest five checkpoints, and we can restore them for future use.

Putting it all together, we obtain the following Python script:

```
# Parameters
learning_rate = 0.01
training_epochs = 1000
batch_size = 100
display_step = 1

with tf.Graph().as_default():

    # mnist data image of shape 28*28=784
    x = tf.placeholder("float", [None, 784])

    # 0-9 digits recognition => 10 classes
    y = tf.placeholder("float", [None, 10])

    output = inference(x)

    cost = loss(output, y)

    global_step = tf.Variable(0, name='global_step',
                              trainable=False)

    train_op = training(cost, global_step)

    eval_op = evaluate(output, y)

    summary_op = tf.merge_all_summaries()

    saver = tf.train.Saver()

    sess = tf.Session()

    summary_writer = tf.train.SummaryWriter("logistic_logs/",
                              graph_def=sess.graph_def)


    init_op = tf.initialize_all_variables()

    sess.run(init_op)


    # Training cycle
    for epoch in range(training_epochs):

        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
```

```
        for i in range(total_batch):
            mbatch_x, mbatch_y = mnist.train.next_batch(
                batch_size)
            # Fit training using batch data
            feed_dict = {x : mbatch_x, y : mbatch_y}
            sess.run(train_op, feed_dict=feed_dict)
            # Compute average loss
            minibatch_cost = sess.run(cost,
                feed_dict=feed_dict)
            avg_cost += minibatch_cost/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            val_feed_dict = {
                 x : mnist.validation.images,
                 y : mnist.validation.labels
            }
            accuracy = sess.run(eval_op,
                feed_dict=val_feed_dict)

            print "Validation Error:", (1 - accuracy)

            summary_str = sess.run(summary_op,
                                   feed_dict=feed_dict)
            summary_writer.add_summary(summary_str,
                                       sess.run(global_step))

            saver.save(sess, "logistic_logs/model-checkpoint",
                       global_step=global_step)


    print "Optimization Finished!"

    test_feed_dict = {
         x : mnist.test.images,
         y : mnist.test.labels
    }

    accuracy = sess.run(eval_op, feed_dict=test_feed_dict)

    print "Test Accuracy:", accuracy
```

Running the script gives us a final accuracy of 91.9% on the test set within 100 epochs of training. This isn't bad, but we'll try to do better in the final section of this chapter, when we approach the problem with a feed-forward neural network.

Leveraging TensorBoard to Visualize Computation Graphs and Learning

Once we set up the logging of summary statistics as described in the previous section, we are ready to visualize the data we've collected. TensorFlow comes with a visualization tool called TensorBoard, which provides an easy-to-use interface for navigating through our summary statistics.[22] Launching TensorBoard is as easy as running:

```
tensorboard --logdir=<absolute_path_to_log_dir>
```

The `logdir` flag should be set to the directory where our `tf.train.SummaryWriter` was configured to serialize our summary statistics. Be sure to pass an absolute path (and not a relative path), because otherwise TensorBoard may not be able to find out logs. If we successfully launch TensorBoard, it should be serving our data at *http://localhost:6006/*, which we can navigate to in our browser.

As shown in Figure 3-5, the first tab contains information on the scalar summaries that we collected. We can observe both the per-minibatch cost and the validation error going down over time.

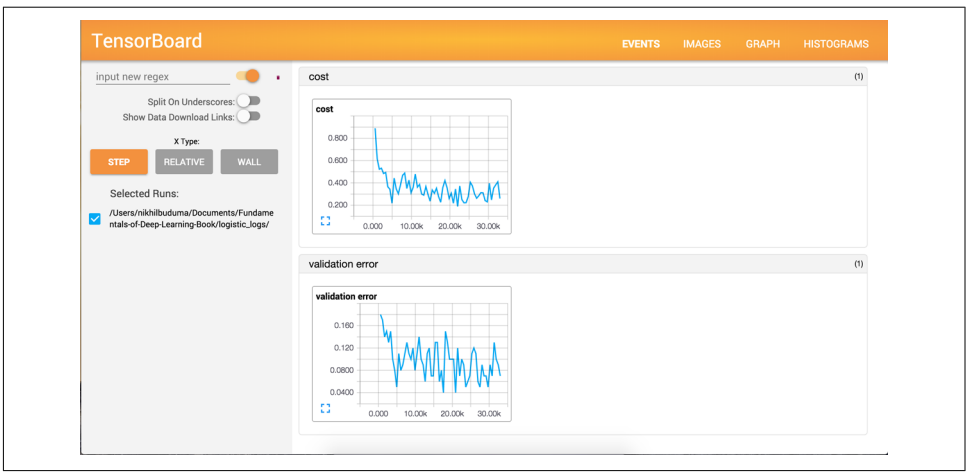


Figure 3-5. The TensorBoard events view

And as Figure 3-6 shows, there's also a tab that allows us to visualize the full computation graph that we've built. It's not particularly easy to interpret, but when we are faced with unexpected behavior, the graph view can serve as a useful debugging tool.

22 *https://www.tensorflow.org/get_started/graph_viz*

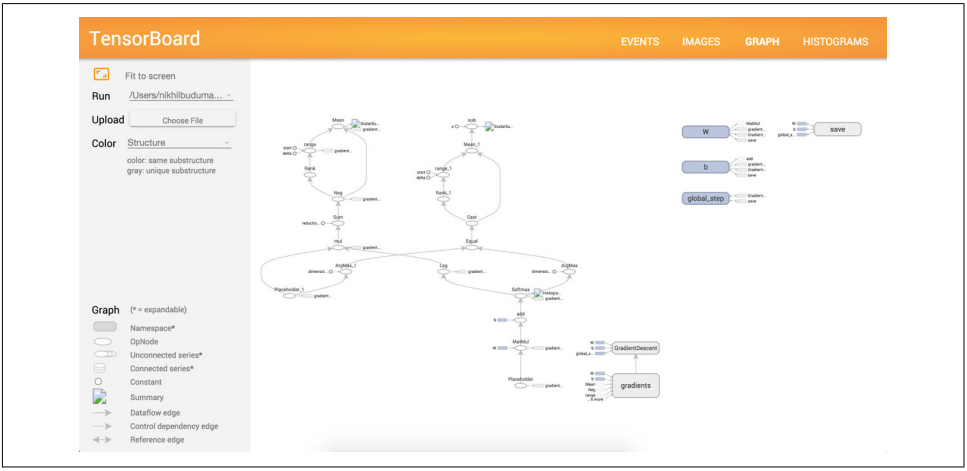


Figure 3-6. The TensorBoard graph view

Building a Multilayer Model for MNIST in TensorFlow

Using a logistic regression model, we were able to achieve an 8.1% error rate on the MNIST dataset. This may seem impressive, but it isn't particularly useful for high-value practical applications. For example, if we were using our system to read personal checks written out for 4-digit amounts ($1,000 to $9,999), we would make errors on nearly 30% of checks! To create an MNIST digit reader that's more practical, let's try to build a feed-forward network to tackle the MNIST challenge.

We construct a feed-forward model with two hidden layers, each with 256 ReLU neurons, as shown in Figure 3-7.

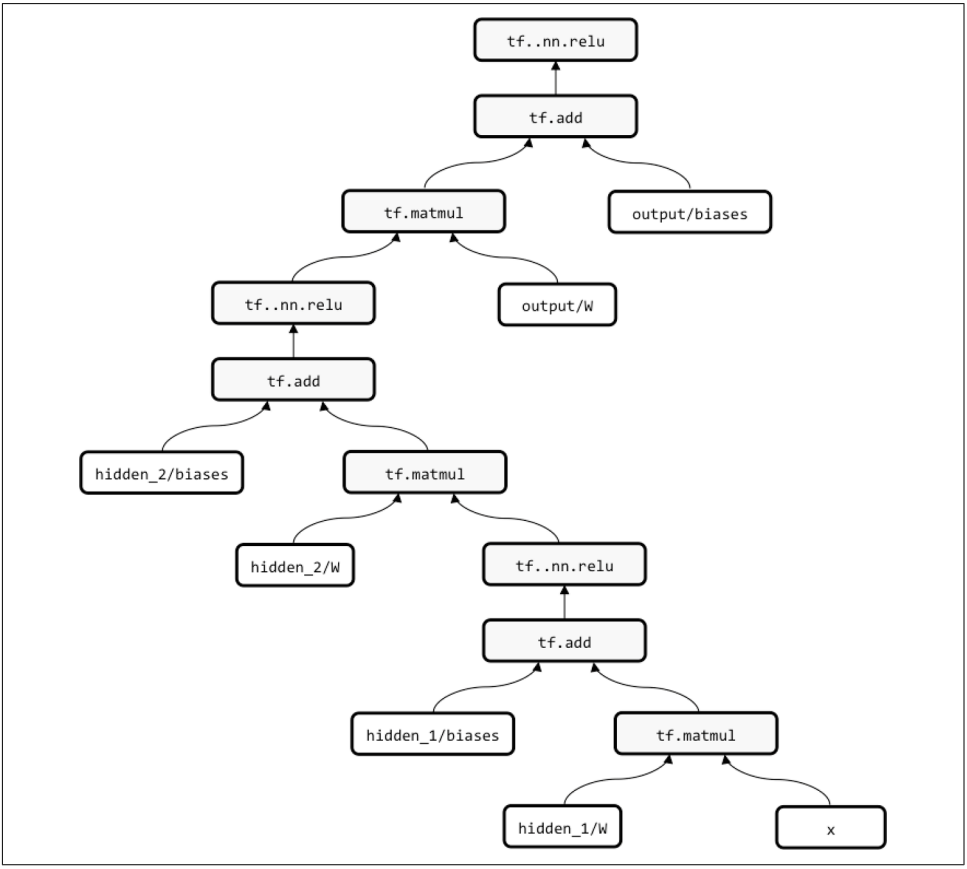


Figure 3-7. A feed-forward network powered by ReLU neurons with two hidden layers

We can reuse most of the code from our logistic regression example with a couple of modifications:

```
def layer(input, weight_shape, bias_shape):
    weight_stddev = (2.0/weight_shape[0])**0.5
    w_init = tf.random_normal_initializer(stddev=weight_stddev)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
                        initializer=w_init)
    b = tf.get_variable("b", bias_shape,
                        initializer=bias_init)
    return tf.nn.relu(tf.matmul(input, W) + b)

def inference(x):
    with tf.variable_scope("hidden_1"):
        hidden_1 = layer(x, [784, 256], [256])

    with tf.variable_scope("hidden_2"):
```

```
        hidden_2 = layer(hidden_1, [256, 256], [256])

    with tf.variable_scope("output"):
        output = layer(hidden_2, [256, 10], [10])

    return output
```

Most of the new code is self explanatory, but our initialization strategy deserves some additional description. The performance of deep neural networks very much depends on an effective initialization of its parameters. As we'll describe in the next chapter, there are many features of the error surfaces of deep neural networks that make optimization using vanilla stochastic gradient descent very difficult. This problem is exacerbated as the number of layers in the model (and thus the complexity of the error surface) increases. Smart initialization is one way to mitigate this issue.

For ReLU units, a study published in 2015 by He et al. demonstrates that the variance of weights in a network should be $\frac{2}{n_{\text{in}}}$, where n_{in} is the number inputs coming into the neuron.[23] The curious reader should investigate what happens when we change our initialization strategy. For example, changing `tf.random_normal_initializer` back to the `tf.random_uniform_initializer` we used in the logistic regression example significantly hurts performance.

Finally, for slightly better performance, we perform the softmax while computing the loss instead of during the inference phase of the network. This results in the following modification:

```
def loss(output, y):
    xentropy = tf.nn.softmax_cross_entropy_with_logits(output, y)
    loss = tf.reduce_mean(xentropy)
    return loss
```

Running this program for 300 epochs gives us a massive improvement over the logistic regression model. The model operates with an accuracy of 98.2%, which is nearly a 78% reduction in the per-digit error rate compared to our first attempt.

23 He, Kaiming, et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *Proceedings of the IEEE International Conference on Computer Vision*. 2015.

Summary

In this chapter, we learned more about using TensorFlow as a library for expressing and training machine learning models. We discussed many critical features of TensorFlow, including management of sessions, variables, operations, computation graphs, and devices. In the final sections, we used this understanding to train and visualize a logistic regression model and a feed-forward neural network using stochastic gradient descent. Although the logistic network model made many errors on the MNIST dataset, our feed-forward network performed much more effectively, making only an average of 1.8 errors out of every 100 digits. We'll improve on this error rate even further in Chapter 5.

In the next chapter, we'll begin to grapple with many of the problems that arise as we start to make our networks deeper. We've already talked about the first piece of the puzzle, which is finding smarter ways to initialize the parameters in our network. In the next chapter, we'll find that as our models become more complex, smart initialization is no longer sufficient for achieving good performance. To overcome these challenges, we'll delve into modern optimization theory and design better algorithms for training deep networks.

CHAPTER 4

Beyond Gradient Descent

The Challenges with Gradient Descent

The fundamental ideas behind neural networks have existed for decades, but it wasn't until recently that neural network-based learning models have become mainstream. Our fascination with neural networks has everything to do with their expressiveness, a quality we've unlocked by creating networks with many layers. As we have discussed in previous chapters, deep neural networks are able to crack problems that were previously deemed intractable. Training deep neural networks end to end, however, is fraught with difficult challenges that took many technological innovations to unravel, including massive labeled datasets (ImageNet, CIFAR, etc.), better hardware in the form of GPU acceleration, and several algorithmic discoveries.

For several years, researchers resorted to layer-wise greedy pre-training in order to grapple with the complex error surfaces presented by deep learning models.[1] These time-intensive strategies would try to find more accurate initializations for the model's parameters one layer at a time before using mini-batch gradient descent to converge to the optimal parameter settings. More recently, however, breakthroughs in optimization methods have enabled us to directly train models in an end-to-end fashion.

In this chapter, we will discuss several of these breakthroughs. The next couple of sections will focus primarily on local minima and whether they pose hurdles for successfully training deep models. In subsequent sections, we will further explore the nonconvex error surfaces induced by deep models, why vanilla mini-batch gradient descent falls short, and how modern nonconvex optimizers overcome these pitfalls.

---

1 Bengio, Yoshua, et al. "Greedy Layer-Wise Training of Deep Networks." *Advances in Neural Information Processing Systems* 19 (2007): 153.

Local Minima in the Error Surfaces of Deep Networks

The primary challenge in optimizing deep learning models is that we are forced to use minimal local information to infer the global structure of the error surface. This is a hard problem because there is usually very little correspondence between local and global structure. Take the following analogy as an example.

Let's assume you're an ant on the continental United States. You're dropped randomly on the map, and your goal is to find the lowest point on this surface. How do you do it? If all you can observe is your immediate surroundings, this seems like an intractable problem. If the surface of the US was bowl shaped (or mathematically speaking, convex) and we were smart about our learning rate, we could use the gradient descent algorithm to eventually find the bottom of the bowl. But the surface of the US is extremely complex, that is to say, is a nonconvex surface, which means that even if we find a valley (a local minimum), we have no idea if it's the lowest valley on the map (the global minimum). In Chapter 2, we talked about how a mini-batch version of gradient descent can help navigate a troublesome error surface when there are spurious regions of magnitude zero gradients. But as we can see in Figure 4-1, even a stochastic error surface won't save us from a deep local minimum.

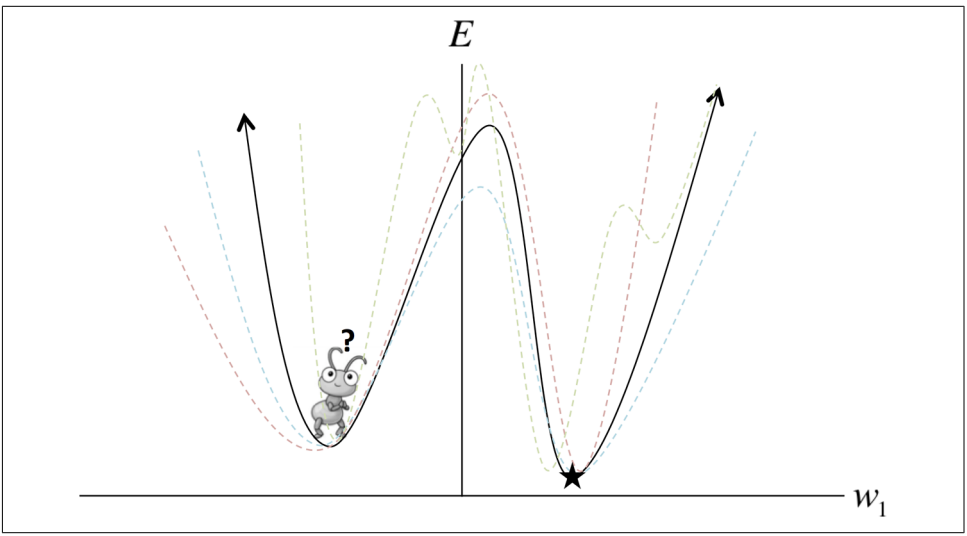


Figure 4-1. Mini-batch gradient descent may aid in escaping shallow local minima, but often fails when dealing with deep local minima, as shown

Now comes the critical question. Theoretically, local minima pose a significant issue. But in practice, how common are local minima in the error surfaces of deep networks? And in which scenarios are they actually problematic for training? In the following two sections, we'll pick apart common misconceptions about local minima.

Model Identifiability

The first source of local minima is tied to a concept commonly referred to as *model identifiability*. One observation about deep neural networks is that their error surfaces are guaranteed to have a large—and in some cases, an infinite—number of local minima. There are two major reasons this observation is true.

The first is that within a layer of a fully-connected feed-forward neural network, any rearrangement of neurons will still give you the same final output at the end of the network. We illustrate this using a simple three-neuron layer in Figure 4-2. As a result, within a layer with n neurons, there are $n!$ ways to rearrange parameters. And for a deep network with l layers, each with n neurons, we have a total of $n!^l$ equivalent configurations.

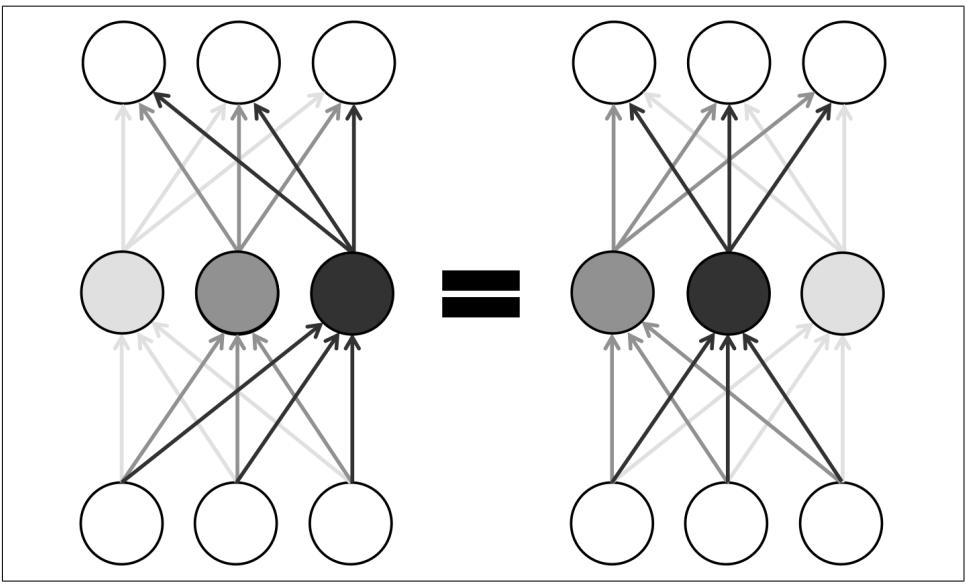


Figure 4-2. Rearranging neurons in a layer of a neural network results in equivalent configurations due to symmetry

In addition to the symmetries of neuron rearrangements, non-identifiability is present in other forms in certain kinds of neural networks. For example, there is an infinite number of equivalent configurations that for an individual ReLU neuron result in equivalent networks. Because an ReLU uses a piecewise linear function, we are free to multiply all of the incoming weights by any nonzero constant k while scaling all of the outgoing weights by $\frac{1}{k}$ without changing the behavior of the network. We leave the justification for this statement as an exercise for the active reader.

Ultimately, however, local minima that arise because of the non-identifiability of deep neural networks are not inherently problematic. This is because all nonidentifiable configurations behave in an indistinguishable fashion no matter what input values they are fed. This means they will achieve the same error on the training, validation, and testing datasets. In other words, all of these models will have learned equally from the training data and will have identical behavior during generalization to unseen examples.

Instead, local minima are only problematic when they are *spurious*. A spurious local minimum corresponds to a configuration of weights in a neural network that incurs a higher error than the configuration at the global minimum. If these kinds of local minima are common, we quickly run into significant problems while using gradient-based optimization methods because we can only take into account local structure.

How Pesky Are Spurious Local Minima in Deep Networks?

For many years, deep learning practitioners blamed all of their troubles in training deep networks on spurious local minima, albeit with little evidence. Today, it remains an open question whether spurious local minima with a high error rate relative to the global minimum are common in practical deep networks. However, many recent studies seem to indicate that most local minima have error rates and generalization characteristics that are very similar to global minima.

One way we might try to naively tackle this problem is by plotting the value of the error function over time as we train a deep neural network. This strategy, however, doesn't give us enough information about the error surface because it is difficult to tell whether the error surface is "bumpy," or whether we merely have a difficult time figuring out which direction we should be moving in.

To more effectively analyze this problem, Goodfellow et al. (a team of researchers collaborating between Google and Stanford) published a paper in 2014 that attempted to separate these two potential confounding factors.[2] Instead of analyzing the error function over time, they cleverly investigated what happens on the error surface between a randomly initialized parameter vector and a successful final solution by using linear interpolation. So given a randomly initialized parameter vector θ_i and stochastic gradient descent (SGD) solution θ_f, we aim to compute the error function at every point along the linear interpolation $\theta_\alpha = \alpha \cdot \theta_f + (1 - \alpha) \cdot \theta_i$.

In other words, they wanted to investigate whether local minima would hinder our gradient-based search method even if we knew which direction to move in. They

2 Goodfellow, Ian J., Oriol Vinyals, and Andrew M. Saxe. "Qualitatively characterizing neural network optimization problems." *arXiv preprint arXiv*:1412.6544 (2014).

showed that for a wide variety of practical networks with different types of neurons, the direct path between a randomly initialized point in the parameter space and a stochastic gradient descent solution isn't plagued with troublesome local minima.

We can even demonstrate this ourselves using the feed-foward ReLU network we built in Chapter 3. Using a checkpoint file that we saved while training our original feed-forward network, we can re-instantiate the `inference` and `loss` components while also maintaining a list of pointers to the variables in the original graph for future use in `var_list_opt` (where `opt` stands for the optimal parameter settings):

```
# mnist data image of shape 28*28=784
x = tf.placeholder("float", [None, 784])
# 0-9 digits recognition => 10 classes
y = tf.placeholder("float", [None, 10])

sess = tf.Session()

with tf.variable_scope("mlp_model") as scope:
    output_opt = inference(x)
    cost_opt = loss(output_opt, y)
    saver = tf.train.Saver()
    scope.reuse_variables()
    var_list_opt = [
        "hidden_1/W",
        "hidden_1/b",
        "hidden_2/W",
        "hidden_2/b",
        "output/W",
        "output/b"
    ]
    var_list_opt = [tf.get_variable(v) for v in var_list_opt]
    saver.restore(sess, "mlp_logs/model-checkpoint-file")
```

Similarly, we can reuse the component constructors to create a randomly initialized network. Here we store the variables in `var_list_rand` for the next step of our program:

```
with tf.variable_scope("mlp_init") as scope:
    output_rand = inference(x)
    cost_rand = loss(output_rand, y)
    scope.reuse_variables()
    var_list_rand = [
        "hidden_1/W",
        "hidden_1/b",
        "hidden_2/W",
        "hidden_2/b",
        "output/W",
        "output/b"
    ]
    var_list_rand = [tf.get_variable(v) for v in var_list_rand]
```

```
    init_op = tf.initialize_variables(var_list_rand)
    sess.run(init_op)
```

With these two networks appropriately initialized, we can now construct the linear interpolation using the mixing parameters `alpha` and `beta`:

```
with tf.variable_scope("mlp_inter") as scope:

    alpha = tf.placeholder("float", [1, 1])
    beta = 1 - alpha

    h1_W_inter = var_list_opt[0] * beta + var_list_rand[0] * alpha
    h1_b_inter = var_list_opt[1] * beta + var_list_rand[1] * alpha
    h2_W_inter = var_list_opt[2] * beta + var_list_rand[2] * alpha
    h2_b_inter = var_list_opt[3] * beta + var_list_rand[3] * alpha
    o_W_inter = var_list_opt[4] * beta + var_list_rand[4] * alpha
    o_b_inter = var_list_opt[5] * beta + var_list_rand[5] * alpha


    h1_inter = tf.nn.relu(tf.matmul(x, h1_W_inter) + h1_b_inter)
    h2_inter = tf.nn.relu(tf.matmul(h1_inter, h2_W_inter) + h2_b_inter)
    o_inter = tf.nn.relu(tf.matmul(h2_inter, o_W_inter) + o_b_inter)


    cost_inter = loss(o_inter, y)
```

Finally, we can vary the value of `alpha` to understand how the error surface changes as we traverse the line between the randomly initialized point and the final SGD solution:

```
import matplotlib.pyplot as plt

summary_writer = tf.train.SummaryWriter("linear_interp_logs/",
                                        graph_def=sess.graph_def)
summary_op = tf.merge_all_summaries()
results = []
for a in np.arange(-2, 2, 0.01):
    feed_dict = {
        x: mnist.test.images,
        y: mnist.test.labels,
        alpha: [[a]],
    }
    cost, summary_str = sess.run([cost_inter, summary_op],
                                 feed_dict=feed_dict)
    summary_writer.add_summary(summary_str, (a + 2)/0.01)
    results.append(cost)


plt.plot(np.arange(-2, 2, 0.01), results, 'ro')
plt.ylabel('Incurred Error')
plt.xlabel('Alpha')
plt.show()
```

This creates Figure 4-3, which we can inspect ourselves. In fact, if we run this experiment over and over again, we find that there are no truly troublesome local minima that would get us stuck. In other words, it seems that the true struggle of gradient descent isn't the existence of troublesome local minima, but instead, is that we have a tough time finding the appropriate direction to move in. We'll return to this thought a little later.

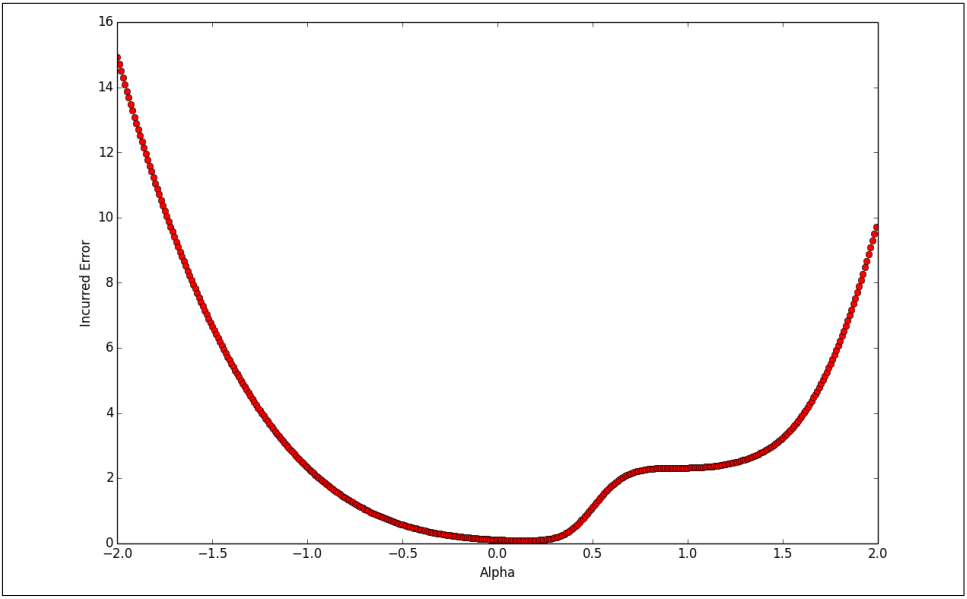


Figure 4-3. The cost function of a three-layer feed-forward network as we linearly interpolate on the line connecting a randomly initialized parameter vector and an SGD solution

Flat Regions in the Error Surface

Although it seems that our analysis is devoid of troublesome local minimum, we do notice a peculiar flat region where the gradient approaches zero when we get to approximately `alpha=1`. This point is not a local minima, so it is unlikely to get us completely stuck, but it seems like the zero gradient might slow down learning if we are unlucky enough to encounter it.

More generally, given an arbitrary function, a point at which the gradient is the zero vector is called a *critical point*. Critical points come in various flavors. We've already talked about local minima. It's also not hard to imagine their counterparts, the *local maxima*, which don't really pose much of an issue for SGD. But then there are these strange critical points that lie somewhere in-between. These "flat" regions that are potentially pesky but not necessarily deadly are called *saddle points*. It turns out that

as our function has more and more dimensions (i.e., we have more and more parameters in our model), saddle points are exponentially more likely than local minima. Let's try to intuit why.

For a one-dimensional cost function, a critical point can take one of three forms, as shown in Figure 4-4. Loosely, let's assume each of these three configurations is equally likely. This means given a random critical point in a random one-dimensional function, it has one-third probability of being a local minimum. This means that if we have a total of k critical points, we can expect to have a total of $\frac{k}{3}$ local minima.

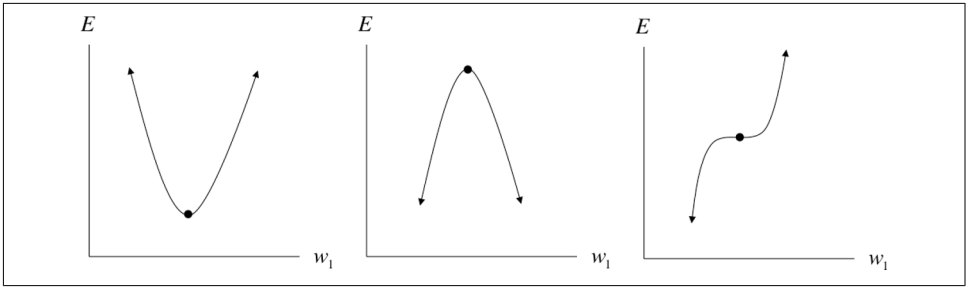


Figure 4-4. Analyzing a critical point along a single dimension

We can also extend this to higher dimensional functions. Consider a cost function operating in a d-dimensional space. Let's take an arbitrary critical point. It turns out that figuring out if this point is a local minimum, local maximum, or a saddle point is a little bit trickier than in the one-dimensional case. Consider the error surface in Figure 4-5. Depending on how you slice the surface (from A to B or from C to D), the critical point looks like either a minimum or a maximum. In reality, it's neither. It's a more complex type of saddle point.

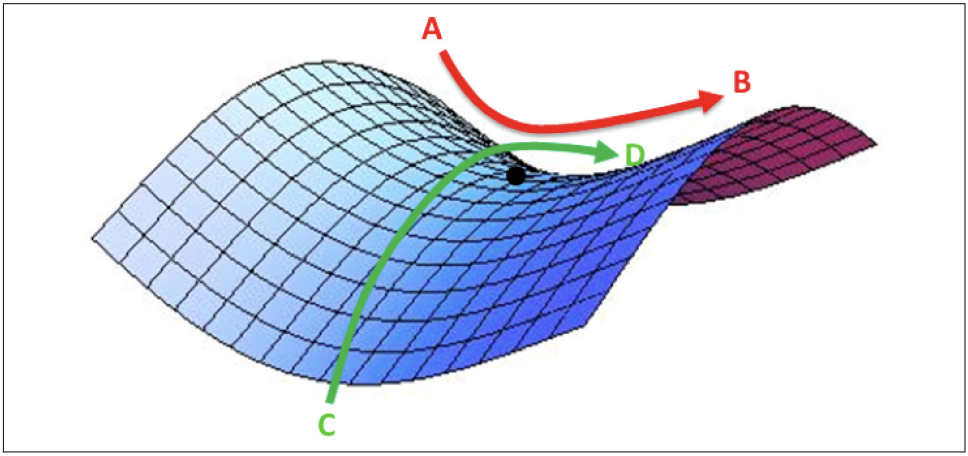


Figure 4-5. A saddle point over a two-dimensional error surface

In general, in a d-dimensional parameter space, we can slice through a critical point on d different axes. A critical point can only be a local minimum if it appears as a local minimum in every single one of the d one-dimensional subspaces. Using the fact that a critical point can come in one of three different flavors in a one-dimensional subspace, we realize that the probability that a random critical point is in a random function is $\frac{1}{3^d}$. This means that a random function function with k critical points has an expected number of $\frac{k}{3^d}$ local minima. In other words, as the dimensionality of our parameter space increases, local minima become exponentially more rare. A more rigorous treatment of this topic is outside the scope of this book, but is explored more extensively by Dauphin et al. in 2014.[3]

So what does this mean for optimizing deep learning models? For stochastic gradient descent, it's still unclear. It seems like these flat segments of the error surface are pesky but ultimately don't prevent stochastic gradient descent from converging to a good answer. However, it does pose serious problems for methods that attempt to directly solve for a point where the gradient is zero. This has been a major hindrance to the usefulness of certain second-order optimization methods for deep learning models, which we will discuss later.

When the Gradient Points in the Wrong Direction

Upon analyzing the error surfaces of deep networks, it seems like the most critical challenge to optimizing deep networks is finding the correct trajectory to move in. It's no surprise, however, that this is a major challenge when we look at what happens to the error surface around a local minimum. As an example, we consider an error surface defined over a two-dimensional parameter space, as shown in Figure 4-6.

3 Dauphin, Yann N., et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization." *Advances in Neural Information Processing Systems*. 2014.

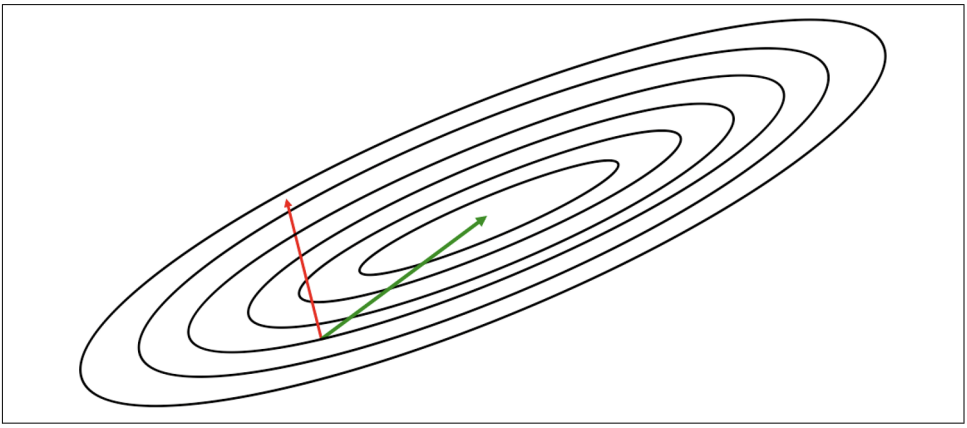


Figure 4-6. Local information encoded by the gradient usually does not corroborate the global structure of the error surface

Revisiting the contour diagrams we explored in Chapter 2, we notice that the gradient isn't usually a very good indicator of the good trajectory. Specifically, we realize that only when the contours are perfectly circular does the gradient always point in the direction of the local minimum. However, if the contours are extremely elliptical (as is usually the case for the error surfaces of deep networks), the gradient can be as inaccurate as 90 degrees away from the correct direction!

We extend this analysis to an arbitrary number of dimensions using some mathematical formalism. For every weight w_i in the parameter space, the gradient computes the value of $\frac{\partial E}{\partial w_i}$, or how the value of the error changes as we change the value of w_i. Taken together over all weights in the parameter space, the gradient gives us the direction of steepest descent. The general problem with taking a significant step in this direction, however, is that the gradient could be changing under our feet as we move! We demonstrate this simple fact in Figure 4-7. Going back to the two-dimensional example, if our contours are perfectly circular and we take a big step in the direction of the steepest descent, the gradient doesn't change direction as we move. However, this is not the case for highly elliptical contours.

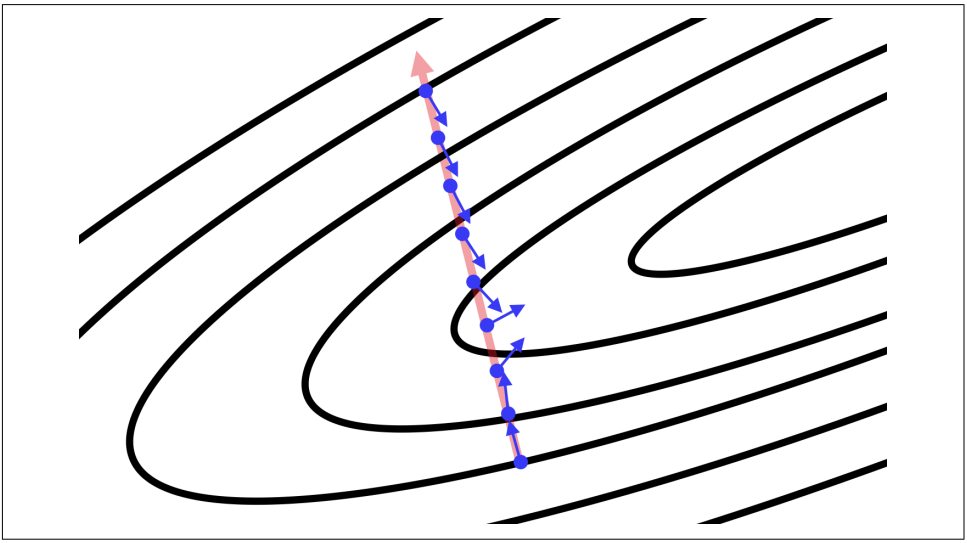


Figure 4-7. We show how the direction of the gradient changes as we move along the direction of steepest descent (as determined from a starting point). The gradient vectors are normalized to identical length to emphasize the change in direction of the gradient vector.

More generally, we can quantify how the gradient changes under our feet as we move in a certain direction by computing second derivatives. Specifically, we want to measure $\frac{\partial\left(\partial E/\partial w_j\right)}{\partial w_i}$, which tells us how the gradient component for w_j changes as we change the value of w_i. We can compile this information into a special matrix known as the *Hessian matrix* ($\boldsymbol{H}$). And when describing an error surface where the gradient changes underneath our feet as we move in the direction of steepest descent, this matrix is said to be *ill-conditioned*.

For the mathematically inclined reader, we go into slightly more detail about how the Hessian limits optimization purely by gradient descent. Certain properties of the Hessian matrix (specifically that it is real and symmetric) allow us to efficiently determine the second derivative (which approximates the curvature of a surface) as we move in a specific direction. Specifically, if we have a unit vector $\boldsymbol{d}$, the second derivative in that direction is given by $\boldsymbol{d}^H\boldsymbol{d}$. We can now use a second-order approximation via Taylor series to understand what happens to the error function as we step from the current parameter vector $\mathbf{x}^{(i)}$ to a new parameter vector $\mathbf{x}$ along gradient vector $\mathbf{g}$ evaluated at $\mathbf{x}^{(i)}$:

$$E(\mathbf{x}) \approx E\left(\mathbf{x}^{(i)}\right) + \left(\mathbf{x} - \mathbf{x}^{(i)}\right)^\top \mathbf{g} + \frac{1}{2}\left(\mathbf{x} - \mathbf{x}^{(i)}\right)^\top \mathbf{H}\left(\mathbf{x} - \mathbf{x}^{(i)}\right)$$

If we go further to state that we will be moving ϵ units in the direction of the gradient, we can further simplify our expression:

$$E\left(\mathbf{x}^{(i)} - \epsilon\mathbf{g}\right) \approx E\left(\mathbf{x}^{(i)}\right) - \epsilon\mathbf{g}^\top\mathbf{g} + \frac{1}{2}\epsilon^2\mathbf{g}^\top\boldsymbol{H}\boldsymbol{g}$$

This expression consists of three terms: 1) the value of the error function at the original parameter vector, 2) the improvement in error afforded by the magnitude of the gradient, and 3) a correction term that incorporates the curvature of the surface as represented by the Hessian matrix.

In general, we should be able to use this information to design better optimization algorithms. For instance, we can even naively take the second order approximation of the error function to determine the learning rate at each step that maximizes the reduction in the error function. It turns out, however, that computing the Hessian matrix exactly is a difficult task. In the next several sections, we'll describe optimization breakthroughs that tackle ill-conditioning without directly computing the Hessian matrix.

Momentum-Based Optimization

Fundamentally, the problem of an ill-conditioned Hessian matrix manifests itself in the form of gradients that fluctuate wildly. As a result, one popular mechanism for dealing with ill-conditioning bypasses the computation of the Hessian, and instead, focuses on how to cancel out these fluctuations over the duration of training.

One way to think about how we might tackle this problem is by investigating how a ball rolls down a hilly surface. Driven by gravity, the ball eventually settles into a minimum on the surface, but for some reason, it doesn't suffer from the wild fluctuations and divergences that happen during gradient descent. Why is this the case? Unlike in stochastic gradient descent (which only uses the gradient), there are two major components that determine how a ball rolls down an error surface. The first, which we already model in SGD as the gradient, is what we commonly refer to as acceleration. But acceleration does not single-handedly determine the ball's movements. Instead, its motion is more directly determined by its velocity. Acceleration only indirectly changes the ball's position by modifying its velocity.

Velocity-driven motion is desirable because it counteracts the effects of a wildly fluctuating gradient by smoothing the ball's trajectory over its history. Velocity serves as a form of memory, and this allows us to more effectively accumulate movement in the direction of the minimum while canceling out oscillating accelerations in orthogonal directions. Our goal, then, is to somehow generate an analog for velocity in our optimization algorithm. We can do this by keeping track of an *exponentially weighted decay* of past gradients. The premise is simple: every update is computed by combining the update in the last iteration with the current gradient. Concretely, we compute the change in the parameter vector as follows:

$$\mathbf{v}_i = m\mathbf{v}_{i-1} - \epsilon\mathbf{g}_i$$

$$\theta_i = \theta_{i-1} + \mathbf{v}_i$$

In other words, we use the momentum hyperparameter m to determine what fraction of the previous velocity to retain in the new update, and add this "memory" of past gradients to our current gradient. This approach is commonly referred to as *momentum*.[4] Because the momentum term increases the step size we take, using momentum may require a reduced learning rate compared to vanilla stochastic gradient descent.

To better visualize how momentum works, we'll explore a toy example. Specifically, we'll investigate how momentum affects updates during a *random walk*. A random walk is a succession of randomly chosen steps. In our example, we'll imagine a particle on a line that, at every time interval, randomly picks a step size between -10 and 10 and takes a moves in that direction. This is simply expressed as:

```
step_range = 10
step_range = 10step_choices = range(-1 * step_range,
                                    step_range + 1)
rand_walk = [random.choice(step_choices) for x in xrange(100)]
```

We'll then simulate what happens when we use a slight modification of momentum (i.e., the standard exponentially weighted moving average algorithm) to smooth our choice of step at every time interval. Again, we can concisely express this as:

```
momentum_rand_walk = [random.choice(step_choices)]
for i in xrange(len(rand_walk) - 1):
    prev = momentum_rand_walk[-1]
    rand_choice = random.choice(step_choices)
    new_step = momentum * prev + (1 - momentum) * rand_choice
    momentum_rand_walk.append()
```

The results, as we vary the momentum from 0 to 1, are quite staggering. Momentum significantly reduces the volatility of updates. The larger the momentum, the less responsive we are to new updates (e.g., a large inaccuracy on the first estimation of trajectory propagates for a significant period of time). We summarize the results of our toy experiment in Figure 4-8.

4 Polyak, Boris T. "Some methods of speeding up the convergence of iteration methods." *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964): 1-17.

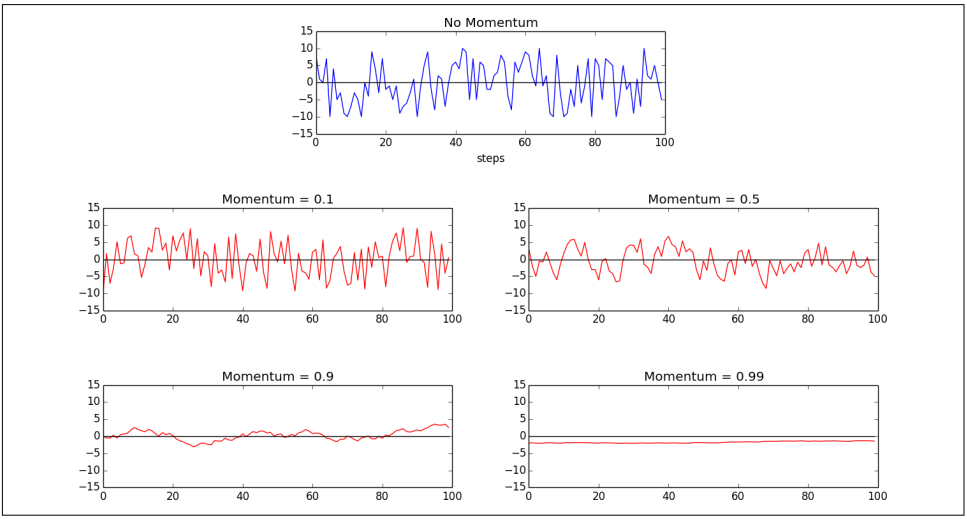


Figure 4-8. Momentum smooths volatility in the step sizes during a random walk using an exponentially weighted moving average

To investigate how momentum actually affects the training of feedforward neural networks, we can retrain our trusty MNIST feedforward network with a TensorFlow momentum optimizer. In this case we can get away with using the same learning rate (0.01) with a typical momentum of 0.9:

```
learning_rate = 0.01
momentum = 0.9
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
train_op = optimizer.minimize(cost, global_step=global_step)
```

The resulting speedup is staggering. We display how the cost function changes over time by comparing the TensorBoard visualizations in Figure 4-9. The figure demonstrates that to achieve a cost of 0.1 without momentum (right) requires nearly 18,000 steps (minibatches), whereas with momentum (left), we require just over 2,000.

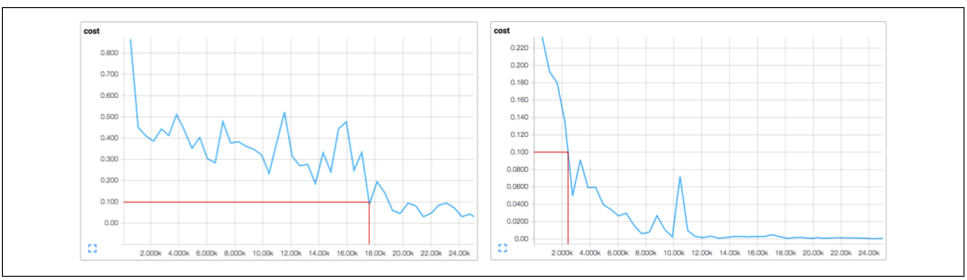


Figure 4-9. Comparing training a feed-forward network with (right) and without (left) momentum demonstrates a massive decrease in training time

Recently, more work has been done exploring how the classical momentum technique can be improved. Sutskever et al. in 2013 proposed an alternative called Nesterov momentum, which computes the gradient on the error surface at $\theta + \mathbf{v}_{i-1}$ during the velocity update instead of at θ.[5] This subtle difference seems to allow Nesterov momentum to change its velocity in a more responsive way. It's been shown that this method has clear benefits in batch gradient descent (convergence guarantees and the ability to use a higher momentum for a given learning rate as compared to classical momentum), but it's not entirely clear whether this is true for the more stochastic mini-batch gradient descent used in most deep learning optimization approaches. Support for Nerestov momentum is not yet available out of the box in TensorFlow as of the writing of this text.

A Brief View of Second-Order Methods

As we discussed in previous sections, computing the Hessian is a computationally difficult task, and momentum afforded us significant speedup without having to worry about it altogether. Several second-order methods, however, have been researched over the past several years that attempt to approximate the Hessian directly. For completeness, we give a broad overview of these methods, but a detailed treatment is beyond the scope of this text.

The first is conjugate gradient descent, which arises out of attempting to improve on a naive method of steepest descent. In steepest descent, we compute the direction of the gradient and then line search to find the minimum along that direction. We jump to the minimum and then recompute the gradient to determine the direction of the next line search. It turns out that this method ends up zigzagging a significant amount, as shown in Figure 4-9, because each time we move in the direction of steepest descent, we undo a little bit of progress in another direction. A remedy to this problem is moving in a *conjugate direction* relative to the previous choice instead of the direction of steepest descent. The conjugate direction is chosen by using an indirect approximation of the Hessian to linearly combine the gradient and our previous direction. With a slight modification, this method generalizes to the nonconvex error surfaces we find in deep networks.[6]

5 Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." *ICML* (3) 28 (2013): 1139-1147.

6 Møller, Martin Fodslette. "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning." *Neural Networks* 6.4 (1993): 525-533.

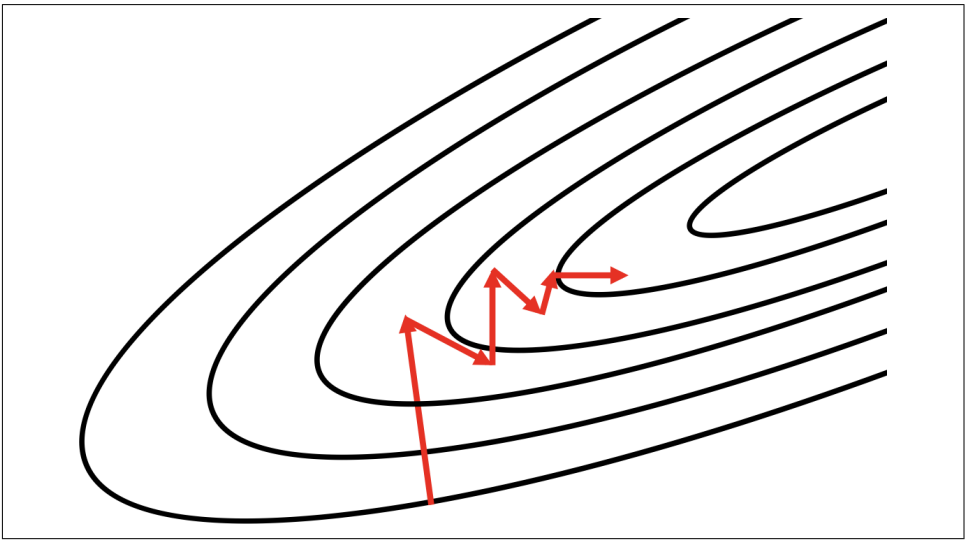


Figure 4-10. The method of steepest descent often zigzags; conjugate descent attempts to remedy this issue

An alternative optimization algorithm known as the *Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm* attempts to compute the inverse of the Hessian matrix iteratively and use the inverse Hessian to more effectively optimize the parameter vector.[7] In its original form, BFGS has a significant memory footprint, but recent work has produced a more memory-efficient version known as *L-BFGS*.[8]

In general, while these methods hold some promise, second-order methods are still an area of active research and are unpopular among practitioners. TensorFlow does not currently support either conjugate gradient descent or L-BFGS at the time of writing this text, although these features seem to be in the development pipeline.

Learning Rate Adaptation

As we have discussed previously, another major challenge for training deep networks is appropriately selecting the learning rate. Choosing the correct learning rate has long been one of the most troublesome aspects of training deep networks because it has a major impact on a network's performance. A learning rate that is too small

7 Broyden, C. G. "A new method of solving nonlinear simultaneous equations." *The Computer Journal* 12.1 (1969): 94-99.

8 Bonnans, Joseph-Frédéric, et al. *Numerical Optimization: Theoretical and Practical Aspects*. Springer Science & Business Media, 2006.

doesn't learn quickly enough, but a learning rate that is too large may have difficulty converging as we approach a local minimum or region that is ill-conditioned.

One of the major breakthroughs in modern deep network optimization was the advent of learning rate adaption. The basic concept behind learning rate adaptation is that the optimal learning rate is appropriately modified over the span of learning to achieve good convergence properties. Over the next several sections, we'll discuss AdaGrad, RMSProp, and Adam, three of the most popular adaptive learning rate algorithms.

AdaGrad—Accumulating Historical Gradients

The first algorithm we'll discuss is AdaGrad, which attempts to adapt the global learning rate over time using an accumulation of the historical gradients, first proposed by Duchi et al. in 2011.[9] Specifically, we keep track of a learning rate for each parameter. This learning rate is inversely scaled with respect to the square root of the sum of the squares (root mean square) of all the parameter's historical gradients.

We can express this mathematically. We initialize a gradient accumulation vector $\mathbf{r}_0 = \mathbf{0}$. At every step, we accumulate the square of all the gradient parameters as follows (where the $\odot$ operation is element-wise tensor multiplication):

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \mathbf{g}_i \odot \mathbf{g}_i$$

Then we compute the update as usual, except our global learning rate ϵ is divided by the square root of the gradient accumulation vector:

$$\theta_i = \theta_{i-1} - \frac{\epsilon}{\delta \oplus \sqrt{\mathbf{r}_i}} \odot \mathbf{g}$$

Note that we add a tiny number δ ($\sim 10^{-7}$) to the denominator in order to prevent division by zero. Also, the division and addition operations are broadcast to the size of the gradient accumulation vector and applied element-wise. In TensorFlow, a built-in optimizer allows for easily utilizing AdaGrad as a learning algorithm:

```
tf.train.AdagradOptimizer(learning_rate,
                          initial_accumulator_value=0.1,
                          use_locking=False,
                          name='Adagrad')
```

The only hitch is that in TensorFlow, the δ and initial gradient accumulation vector are rolled together into the `initial_accumulator_value` argument.

9 Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12.Jul (2011): 2121-2159.

On a functional level, this update mechanism means that the parameters with the largest gradients experience a rapid decrease in their learning rates, while parameters with smaller gradients only observe a small decrease in their learning rates. The ultimate effect is that AdaGrad forces more progress in the more gently sloped directions on the error surface, which can help overcome ill-conditioned surfaces. This results in some good theoretical properties, but in practice, training deep learning models with AdaGrad can be somewhat problematic. Empirically, AdaGrad has a tendency to cause a premature drop in learning rate, and as a result doesn't work particularly well for some deep models. In the next section, we'll describe RMSProp, which attempts to remedy this shortcoming.

RMSProp—Exponentially Weighted Moving Average of Gradients

While AdaGrad works well for simple convex functions, it isn't designed to navigate the complex error surfaces of deep networks. Flat regions may force AdaGrad to decrease the learning rate before it reaches a minimum. The conclusion is that simply using a naive accumulation of gradients isn't sufficient.

Our solution is to bring back a concept we introduced earlier while discussing momentum to dampen fluctuations in the gradient. Compared to naive accumulation, exponentially weighted moving averages also enable us to "toss out" measurements that we made a long time ago. More specifically, our update to the gradient accumulation vector is now as follows:

$$\mathbf{r}_i = \rho \mathbf{r}_{i-1} + (1 - \rho)\mathbf{g}_i \odot \mathbf{g}_i$$

The decay factor ρ determines how long we keep old gradients. The smaller the decay factor, the shorter the effective window. Plugging this modification into AdaGrad gives rise to the RMSProp learning algorithm, first proposed by Geoffrey Hinton.[10]

In TensorFlow, we can instantiate the RMSProp optimizer with the following code. We note that in this case, unlike in Adagrad, we pass in δ separately as the `epsilon` argument to the constructor:

```
tf.train.RMSPropOptimizer(learning_rate, decay=0.9,
                          momentum=0.0, epsilon=1e-10,
                          use_locking=False, name='RMSProp')
```

As the template suggests, we can utilize RMSProp with momentum (specifically Nerestov momentum). Overall, RMSProp has been shown to be a highly effective optimizer for deep neural networks, and is a default choice for many seasoned practitioners.

---

10 Tieleman, Tijmen, and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." *COURSERA: Neural Networks for Machine Learning* 4.2 (2012).

Adam—Combining Momentum and RMSProp

Before concluding our discussion of modern optimizers, we discuss one final algorithm—Adam.[11] Spiritually, we can think about Adam as a variant combination of RMSProp and momentum.

The basic idea is as follows. We want to keep track of an exponentially weighted moving average of the gradient (essentially the concept of velocity in classical momentum), which we can express as follows:

$$\mathbf{m}_i = \beta_1 \mathbf{m}_{i-1} + (1-\beta_1)\mathbf{g}_i$$

This is our approximation of what we call the *first moment* of the gradient, or $\mathbb{E}[\mathbf{g}_i]$. And similarly to RMSProp, we can maintain an exponentially weighted moving average of the historical gradients. This is our estimation of what we call the *second moment* of the gradient, or $\mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i]$:

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1-\beta_2)\mathbf{g}_i \odot \mathbf{g}_i$$

However, it turns out these estimations are biased relative to the real moments because we start off by initializing both vectors to the zero vector. In order to remedy this bias, we derive a correction factor for both estimations. Here, we describe the derivation for the estimation of the second moment. The derivation for the first moment, which is analogous to the derivation here, is left as an exercise for the mathematically inclined reader.

We begin by expressing the estimation of the second moment in terms of all past gradients. This is done by simply expanding the recurrence relationship:

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1-\beta_2)\mathbf{g}_i \odot \mathbf{g}_i$$

$$\mathbf{v}_i = \beta_2^{i-1}(1-\beta_2)\mathbf{g}_1 \odot \mathbf{g}_1 + \beta_2^{i-2}(1-\beta_2)\mathbf{g}_2 \odot \mathbf{g}_2 + \ldots + (1-\beta_2)\mathbf{g}_i \odot \mathbf{g}_i$$

$$\mathbf{v}_i = (1-\beta_2)\Sigma_{k=1}^{i}\beta^{i-k}\mathbf{g}_k \odot \mathbf{g}_k$$

We can then take the expected value of both sides to determine how our estimation $\mathbb{E}[\mathbf{v}_i]$ compares to the real value of $\mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i]$:

$$\mathbb{E}[\mathbf{v}_i] = \mathbb{E}\left[(1-\beta_2)\Sigma_{k=1}^{i}\beta^{i-k}\mathbf{g}_k \odot \mathbf{g}_k\right]$$

We can also assume that $\mathbb{E}[\mathbf{g}_k \odot \mathbf{g}_k] \approx \mathbb{E}[\mathbf{g}_i \approx \mathbf{g}_i]$, because even if the second moment of the gradient has changed since a historical value, β_2 should be chosen so that the old second moments of the gradients are essentially decayed out of relevancy. As a result, we can make the following simplification:

---

11 Kingma, Diederik, and Jimmy Ba. "Adam: A Method for Stochastic Optimization." *arXiv preprint arXiv*: 1412.6980 (2014).

$$\mathbb{E}[\mathbf{v}_i] \approx \mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i](1-\beta_2)\Sigma_{k=1}^{i}\beta^{i-k}$$

$$\mathbb{E}[\mathbf{v}_i] \approx \mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i](1\text{-}\beta_2{}^{i})$$

Note that we make the final simplification using the elementary algebraic identity $1-x^n = (1-x)\left(1+x+...+x^{n-1}\right)$. The results of this derivation and the analogous derivation for the first moment are the following correction schemes to account for the initialization bias:

$$\tilde{\mathbf{m}}_i = \frac{m_i}{1-\beta_1^i}$$

$$\tilde{\mathbf{v}}_i = \frac{\tilde{\mathbf{v}}_i}{1-\beta_2^i}$$

We can then use these corrected moments to update the parameter vector, resulting in the final Adam update:

$$\theta_i = \theta_{i-1} - \frac{\epsilon}{\delta \oplus \sqrt{\tilde{\mathbf{v}}_i}}\tilde{\mathbf{m}}_i$$

Recently, Adam has gained popularity because of its corrective measures against the zero initialization bias (a weakness of RMSProp) and its ability to combine the core concepts behind RMSProp with momentum more effectively. TensorFlow exposes the Adam optimizer through the following constructor:

```
tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
                       beta2=0.999, epsilon=1e-08,
                       use_locking=False, name='Adam')
```

The default hyperparameter settings for Adam for TensorFlow generally perform quite well, but Adam is also generally robust to choices in hyperparameters. The only exception is that the learning rate may need to be modified in certain cases from the default value of 0.001.

The Philosophy Behind Optimizer Selection

In this chapter, we've discussed several strategies that are used to make navigating the complex error surfaces of deep networks more tractable. These strategies have culminated in several optimization algorithms, each with its own benefits and shortcomings.

While it would be awfully nice to know when to use which algorithm, there is very little consensus among expert practitioners. Currently, the most popular algorithms are mini-batch gradient descent, mini-batch gradient with momentum, RMSProp, RMSProp with momentum, Adam, and AdaDelta (which we haven't discussed here, and is not currently supported by TensorFlow as of the writing of this text). We include a TensorFlow script in the Github repository for this text for the curious reader to experiment with these optimization algorithms on the feed-forward network model we built:

```
$ python optimzer_mlp.py <sgd, momentum, adagrad, rmsprop,
   adam>
```

One important point, however, is that for most deep learning practitioners, the best way to push the cutting edge of deep learning is not by building more advanced optimizers. Instead, the vast majority of breakthroughs in deep learning over the past several decades have been obtained by discovering architectures that are easier to train instead of trying to wrangle with nasty error surfaces. We'll begin focusing on how to leverage architecture to more effectively train neural networks in the rest of this book.

Summary

In this chapter, we discussed several challenges that arise when trying to train deep networks with complex error surfaces. We discussed how while the challenges of spurious local minima may likely be exaggerated, saddle points and ill-conditioning do pose a serious threat to the success of vanilla mini-batch gradient descent. We described how momentum can be used to overcome ill-conditioning, and briefly discussed recent research in second-order methods to approximate the Hessian matrix. We also described the evolution of adaptive learning rate optimizers, which tune the learning rate during the training process for better convergence.

In the next chapter, we'll begin tackling the larger issue of network architecture and design. We'll begin by exploring computer vision and how we might design deep networks that learn effectively from complex images.

CHAPTER 5
Convolutional Neural Networks

Neurons in Human Vision

The human sense of vision is unbelievably advanced. Within fractions of seconds, we can identify objects within our field of view, without thought or hesitation. Not only can we name objects we are looking at, we can also perceive their depth, perfectly distinguish their contours, and separate the objects from their backgrounds. Somehow our eyes take in raw voxels of color data, but our brain transforms that information into more meaningful primitives—lines, curves, and shapes—that might indicate, for example, that we're looking at a house cat.[1]

Foundational to the human sense of vision is the neuron. Specialized neurons are responsible for capturing light information in the human eye.[2] This light information is then preprocessed, transported to the visual cortex of the brain, and then finally analyzed to completion. Neurons are single-handedly responsible for all of these functions. As a result, intuitively, it would make a lot of sense to extend our neural network models to build better computer vision systems. In this chapter, we will use our understanding of human vision to build effective deep learning models for image problems. But before we jump in, let's take a look at more traditional approaches to image analysis and why they fall short.

1 Hubel, David H., and Torsten N. Wiesel. "Receptive fields and functional architecture of monkey striate cortex." *The Journal of Physiology* 195.1 (1968): 215-243.

2 Cohen, Adolph I. "Rods and Cones." *Physiology of Photoreceptor Organs*. Springer Berlin Heidelberg, 1972. 63-110.

The Shortcomings of Feature Selection

Let's begin by considering a simple computer vision problem. I give you a randomly selected image, such as the one in Figure 5-1. Your task is to tell me if there is a human face in this picture. This is exactly the problem that Paul Viola and Michael Jones tackled in their seminal paper published in 2001.[3]

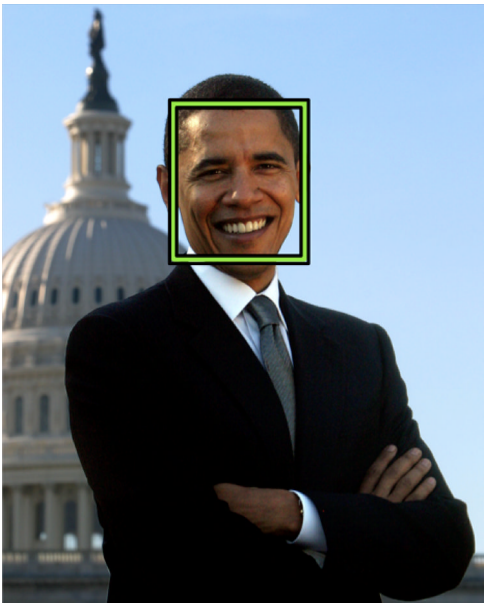


Figure 5-1. A hypothetical face-recognition algorithm should detect a face in this photograph of former President Barack Obama

For a human like you or me, this task is completely trivial. For a computer, however, this is a very difficult problem. How do we teach a computer that an image contains a face? We could try to train a traditional machine learning algorithm (like the one we described in the Chapter 1) by giving it the raw pixel values of the image and hoping it can find an appropriate classifier. Turns out this doesn't work very well at all because the signal-to-noise ratio is much too low for any useful learning to occur. We need an alternative.

The compromise that was eventually reached was essentially a trade-off between the traditional computer program, where the human defined all of the logic, and a pure

3 Viola, Paul, and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features." Computer Vision and Pattern Recognition, 2001. CVPR 2001. *Proceedings of the 2001 IEEE Computer Society Conference* on. Vol. 1. IEEE, 2001.

machine learning approach, where the computer did all of the heavy lifting. In this compromise, a human would choose the features (perhaps hundreds or thousands) that he or she believed were important in making a classification decision. In doing so, the human would be producing a lower-dimensional representation of the same learning problem. The machine learning algorithm would then use these new *feature vectors* to make classification decisions. Because the *feature extraction* process improves the signal-to-noise ratio (assuming the appropriate features are picked), this approach had quite a bit of success compared to the state of the art at the time.

Viola and Jones had the insight that faces had certain patterns of light and dark patches that they could exploit. For example, there is a difference in light intensity between the eye region and the upper cheeks. There is also a difference in light intensity between the nose bridge and the two eyes on either side. These detectors are shown in Figure 5-2.

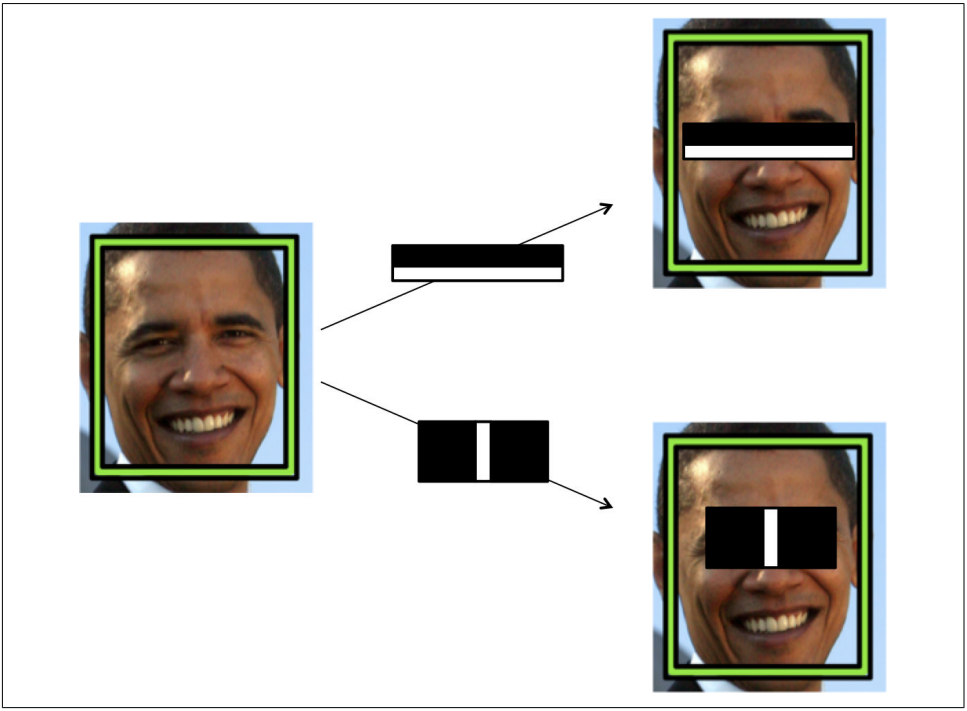


Figure 5-2. An illustration of Viola-Jones intensity detectors

By themselves, each of these features is not very effective at identifying a face. But when used together (through a classic machine learning algorithm known as boosting, described in the original manuscript), their combined effectiveness drastically increases. On a dataset of 130 images and 507 faces, the algorithm achieves a 91.4% detection rate with 50 false positives. The performance was unparalleled at the time,

but there are fundamental limitations of the algorithm. If a face is partially covered with shade, the light intensity comparisons no longer work. Moreover, if the algorithm is looking at a face on a crumpled flier or the face of a cartoon character, it would most likely fail.

The problem is the algorithm hasn't really learned that much about what it means to "see" a face. Beyond differences in light intensity, our brain uses a vast number of visual cues to realize that our field of view contains a human face, including contours, relative positioning of facial features, and color. And even if there are slight discrepancies in one of our visual cues (for example, if parts of the face are blocked from view or if shade modifies light intensities), our visual cortex can still reliably identify faces.

In order to use traditional machine learning techniques to teach a computer to "see," we need to provide our program with a lot more features to make accurate decisions. Before the advent of deep learning, huge teams of computer vision researchers would take years to debate about the usefulness of different features. As the recognition problems became more and more intricate, researchers had a difficult time coping with the increase in complexity.

To illustrate the power of deep learning, consider the ImageNet challenge, one of the most prestigious benchmarks in computer vision (sometimes even referred to as the Olympics of computer vision).[4] Every year, researchers attempt to classify images into one of 200 possible classes given a training dataset of approximately 450,000 images. The algorithm is given five guesses to get the right answer before it moves onto the next image in the test dataset. The goal of the competition is to push the state of the art in computer vision to rival the accuracy of human vision itself (approximately 95–96%). In 2011, the winner of the ImageNet benchmark had an error rate of 25.7%, making a mistake on one out of every four images.[5] Definitely a huge improvement over random guessing, but not good enough for any sort of commercial application. Then in 2012, Alex Krizhevsky from Geoffrey Hinton's lab at the University of Toronto did the unthinkable. Pioneering a deep learning architecture known as a *convolutional neural network* for the first time on a challenge of this size and complexity, he blew the competition out of the water. The runner up in the competition scored a commendable 26.1% error rate. But AlexNet, over the course of just a few months of work, completely crushed 50 years of traditional computer vision research with an

4 Deng, Jia, et al. "ImageNet: A Large-Scale Hierarchical Image Database." *Computer Vision and Pattern Recognition*, 2009. CVPR 2009. IEEE Conference. IEEE, 2009.

5 Perronnin, Florent, Jorge Sénchez, and Yan Liu Xerox. "Large-scale image categorization with explicit data embedding." *Computer Vision and Pattern Recognition* (CVPR), 2010 IEEE Conference. IEEE, 2010.

error rate of approximately 16%.[6] It would be no understatement to say that AlexNet single-handedly put deep learning on the map for computer vision, and completely revolutionized the field.

Vanilla Deep Neural Networks Don't Scale

The fundamental goal in applying deep learning to computer vision is to remove the cumbersome, and ultimately limiting, feature selection process. As we discussed in Chapter 1, deep neural networks are perfect for this process because each layer of a neural network is responsible for learning and building up features to represent the input data that it receives. A naive approach might be for us to use a vanilla deep neural network using the network layer primitive we designed in Chapter 3 for the MNIST dataset to achieve the image classification task.

If we attempt to tackle the image classification problem in this way, however, we'll quickly face a pretty daunting challenge, visually demonstrated in Figure 5-3. In MNIST, our images were only 28 x 28 pixels and were black and white. As a result, a neuron in a fully connected hidden layer would have 784 incoming weights. This seems pretty tractable for the MNIST task, and our vanilla neural net performed quite well. This technique, however, does not scale well as our images grow larger. For example, for a full-color 200 x 200 pixel image, our input layer would have 200 x 200 x 3 = 120,000 weights. And we're going to want to have lots of these neurons over multiple layers, so these parameters add up quite quickly! Clearly, this full connectivity is not only wasteful, but also means that we're much more likely to overfit to the training dataset.

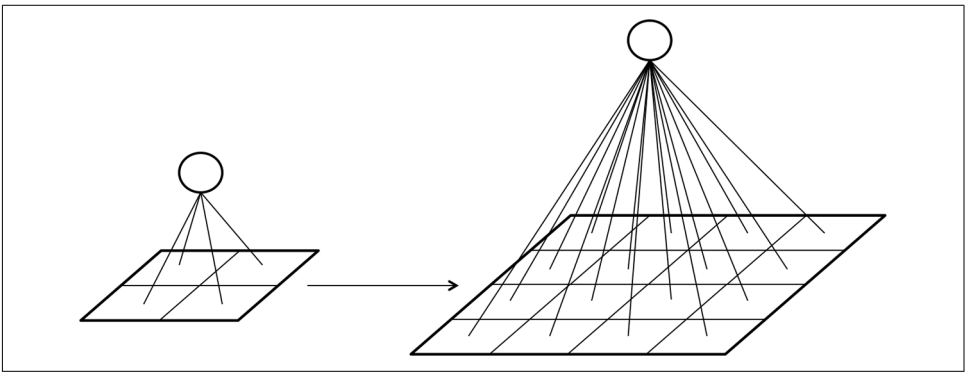


Figure 5-3. The density of connections between layers increases intractably as the size of the image increases

6 Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. 2012.

The convolutional network takes advantage of the fact that we're analyzing images, and sensibly constrains the architecture of the deep network so that we drastically reduce the number of parameters in our model. Inspired by how human vision works, layers of a convolutional network have neurons arranged in three dimensions, so layers have a width, height, and depth, as shown in Figure 5-4.[7] As we'll see, the neurons in a convolutional layer are only connected to a small, local region of the preceding layer, so we avoid the wastefulness of fully-connected neurons. A convolutional layer's function can be expressed simply: it processes a three-dimensional volume of information to produce a new three-dimensional volume of information. We'll take a closer look at how this works in the next section.

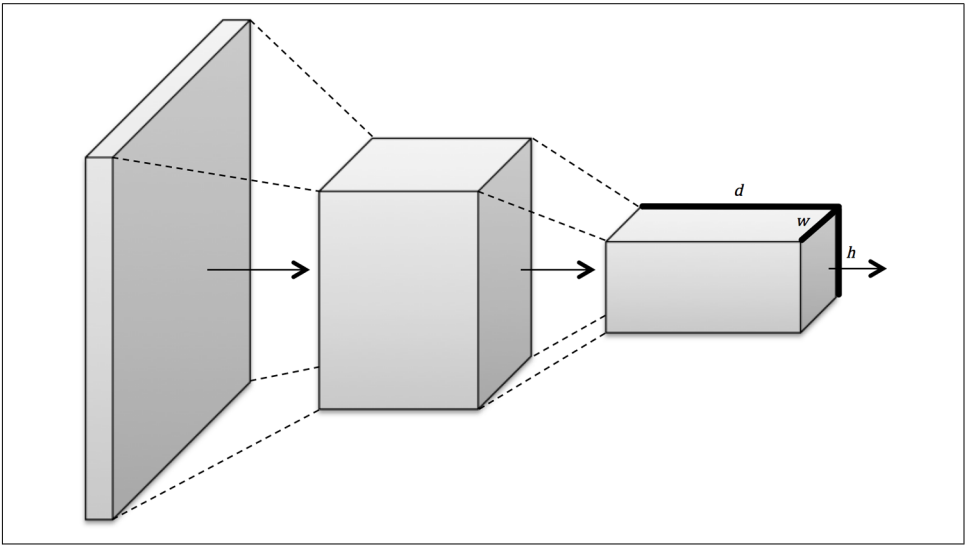


Figure 5-4. Convolutional layers arrange neurons in three dimensions, so layers have width, height, and depth

Filters and Feature Maps

In order to motivate the primitives of the convolutional layer, let's build an intuition for how the human brain pieces together raw visual information into an understanding of the world around us. One of the most influential studies in this space came from David Hubel and Torsten Wiesel, who discovered that parts of the visual cortex are responsible for detecting edges. In 1959, they inserted electrodes into the brain of a cat and projected black-and-white patterns on the screen. They found that some

7 LeCun, Yann, et al. "Handwritten Digit Recognition with a Back-Propagation Network." *Advances in Neural Information Processing Systems*. 1990.

neurons fired only when there were vertical lines, others when there were horizontal lines, and still others when the lines were at particular angles.[8]

Further work determined that the visual cortex was organized in layers. Each layer is responsible for building on the features detected in the previous layers—from lines, to contours, to shapes, to entire objects. Furthermore, within a layer of the visual cortex, the same feature detectors were replicated over the whole area in order to detect features in all parts of an image. These ideas significantly impacted the design of convolutional neural nets.

The first concept that arose was that of a *filter*, and it turns out that here, Viola and Jones were actually pretty close. A filter is essentially a feature detector, and to understand how it works, let's consider the toy image in Figure 5-5.

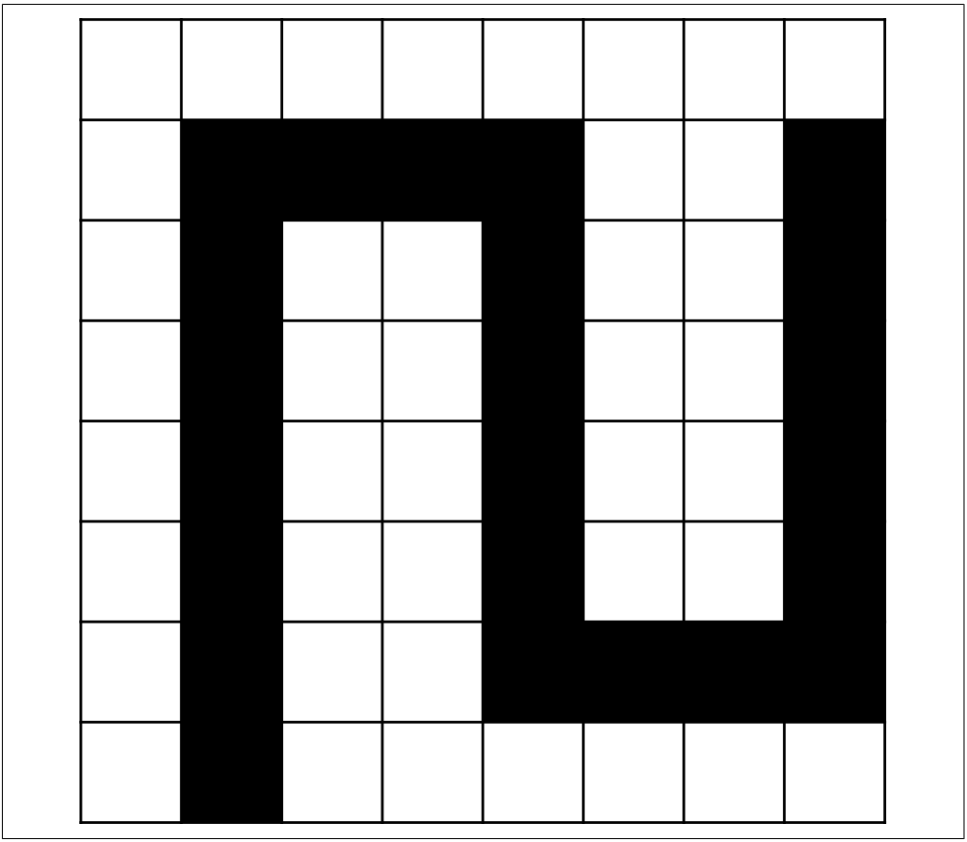


Figure 5-5. We'll analyze this simple black-and-white image as a toy example

8 Hubel, David H., and Torsten N. Wiesel. "Receptive fields of single neurones in the cat's striate cortex." *The Journal of Physiology* 148.3 (1959): 574-591.

Let's say that we want to detect vertical and horizontal lines in the image. One approach would be to use an appropriate feature detector, as shown in Figure 5-6. For example, to detect vertical lines, we would use the feature detector on the top, slide it across the entirety of the image, and at every step check if we have a match. We keep track of our answers in the matrix in the top right. If there's a match, we shade the appropriate box black. If there isn't, we leave it white. This result is our *feature map*, and it indicates where we've found the feature we're looking for in the original image. We can do the same for the horizontal line detector (bottom), resulting in the feature map in the bottom-right corner.

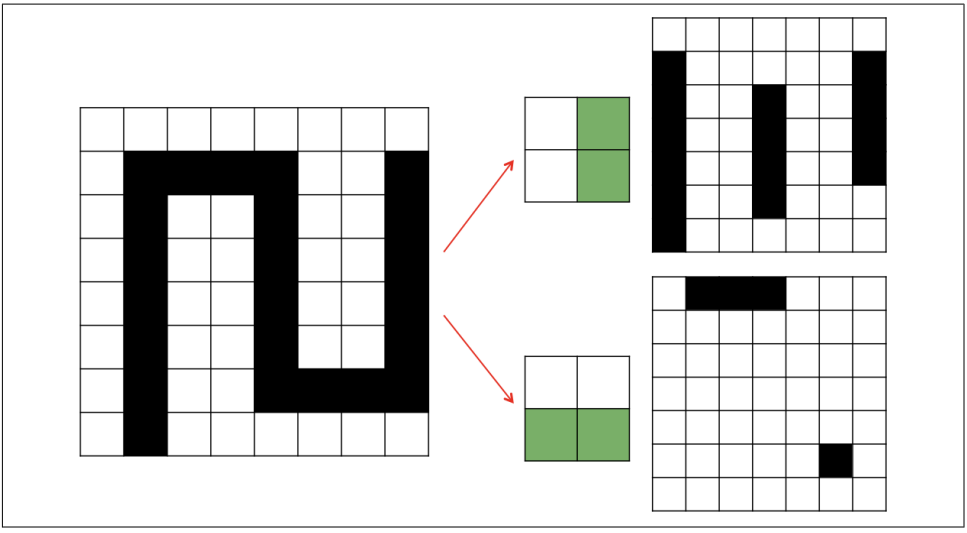


Figure 5-6. Applying filters that detect vertical and horizontal lines on our toy example

This operation is called a convolution. We take a filter and we multiply it over the entire area of an input image. Using the following scheme, let's try to express this operation as neurons in a network. In this scheme, layers of neurons in a feed-forward neural net represent either the original image or a feature map. Filters represent combinations of connections (one such combination is highlighted in Figure 5-7) that get replicated across the entirety of the input. In Figure 5-7, connections of the same color are restricted to always have the same weight. We can achieve this by initializing all the connections in a group with identical weights and by always averaging the weight updates of a group before applying them at the end of each iteration of backpropagation. The output layer is the feature map generated by this filter. A neuron in the feature map is activated if the filter contributing to its activity detected an appropriate feature at the corresponding position in the previous layer.

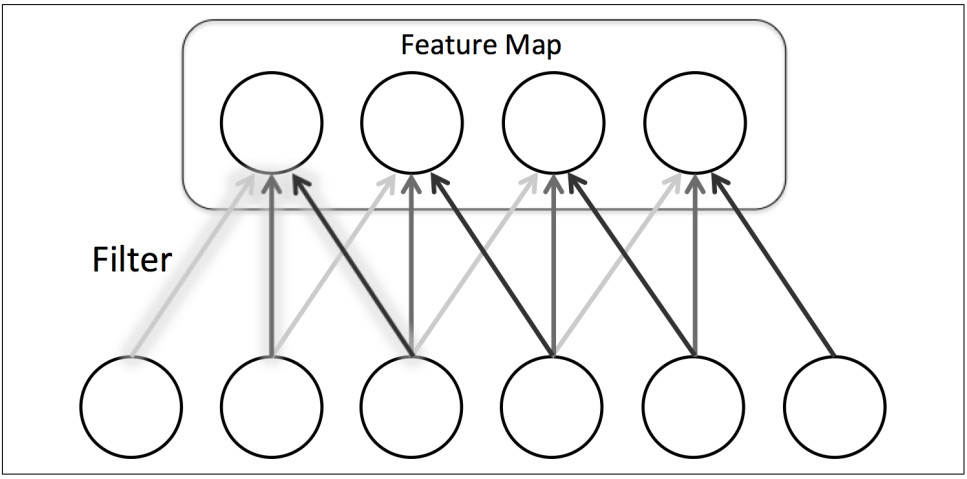


Figure 5-7. Representing filters and feature maps as neurons in a convolutional layer

Let's denote the k^{th} feature map in layer m as m^k. Moreover, let's denote the corresponding filter by the values of its weights W. Then assuming the neurons in the feature map have bias b^k (note that the bias is kept identical for all of the neurons in a feature map), we can mathematically express the feature map as follows:

$$m_{ij}^k = f\left((W * x)_{ij} + b^k\right)$$

This mathematical description is simple and succinct, but it doesn't completely describe filters as they are used in convolutional neural networks. Specifically, filters don't just operate on a single feature map. They operate on the entire volume of feature maps that have been generated at a particular layer. For example, consider a situation in which we would like to detect a face at a particular layer of a convolutional net. And we have accumulated three feature maps, one for eyes, one for noses, and one for mouths. We know that a particular location contains a face if the corresponding locations in the primitive feature maps contain the appropriate features (two eyes, a nose, and a mouth). In other words, to make decisions about the existence of a face, we must combine evidence over multiple feature maps. This is equally necessary for an input image that is of full color. These images have pixels represented as RGB values, and so we require three slices in the input volume (one slice for each color). As a result, feature maps must be able to operate over volumes, not just areas. This is shown below in Figure 5-8. Each cell in the input volume is a neuron. A local portion is multiplied with a filter (corresponding to weights in the convolutional layer) to produce a neuron in a filter map in the following volumetric layer of neurons.

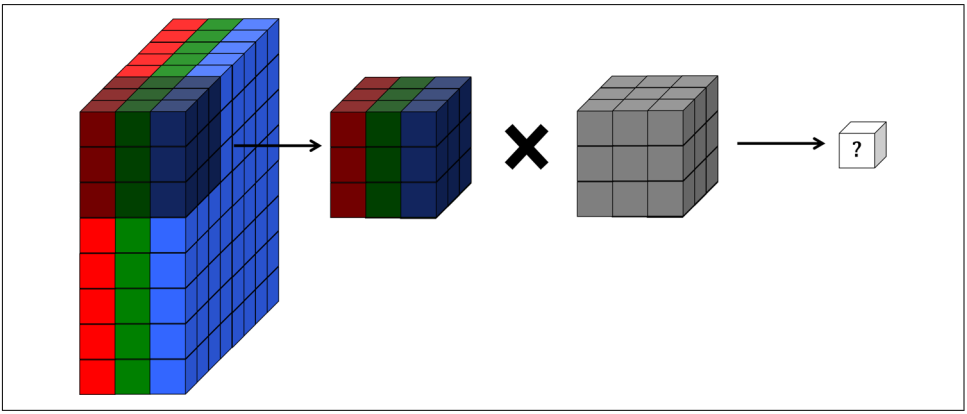


Figure 5-8. Representing a full-color RGB image as a volume and applying a volumetric convolutional filter

As we discussed in the previous section, a convolutional layer (which consists of a set of filters) converts one volume of values into another volume of values. The depth of the filter corresponds to the depth of the input volume. This is so that the filter can combine information from all the features that have been learned. The depth of the output volume of a convolutional layer is equivalent to the number of filters in that layer, because each filter produces its own slice. We visualize these relationships in Figure 5-9.

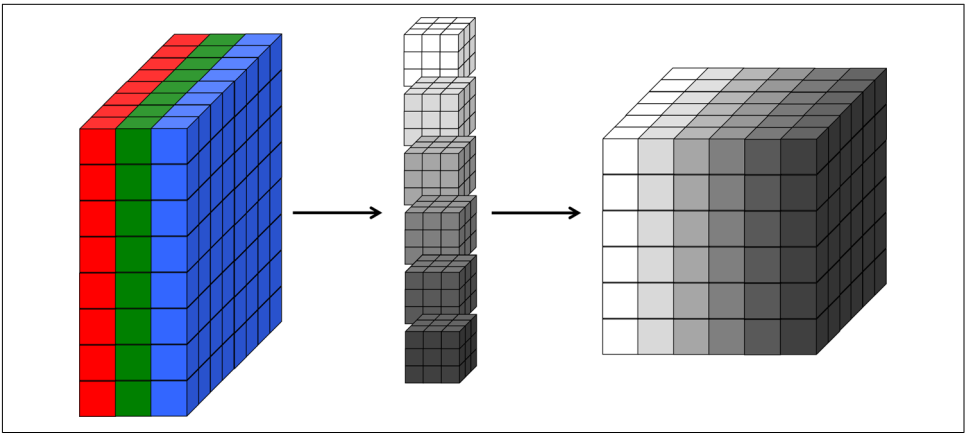


Figure 5-9. A three-dimensional visualization of a convolutional layer, where each filter corresponds to a slice in the resulting output volume

In the next section, we will use these concepts and fill in some of the gaps to create a full description of a convolutional layer.

Full Description of the Convolutional Layer

Let's use the concepts we've developed so far to complete the description of the convolutional layer. First, a convolutional layer takes in an input volume. This input volume has the following characteristics:

- Its *width* w_{in}
- Its *height* h_{in}
- Its *depth* d_{in}
- Its *zero padding* p

This volume is processed by a total of k filters, which represent the weights and connections in the convolutional network. These filters have a number of hyperparameters, which are described as follows:

- Their *spatial extent* e, which is equal to the filter's height and width.
- Their *stride* s, or the distance between consecutive applications of the filter on the input volume. If we use a stride of 1, we get the full convolution described in the previous section. We illustrate this in Figure 5-10.
- The bias b (a parameter learned like the values in the filter) which is added to each component of the convolution.

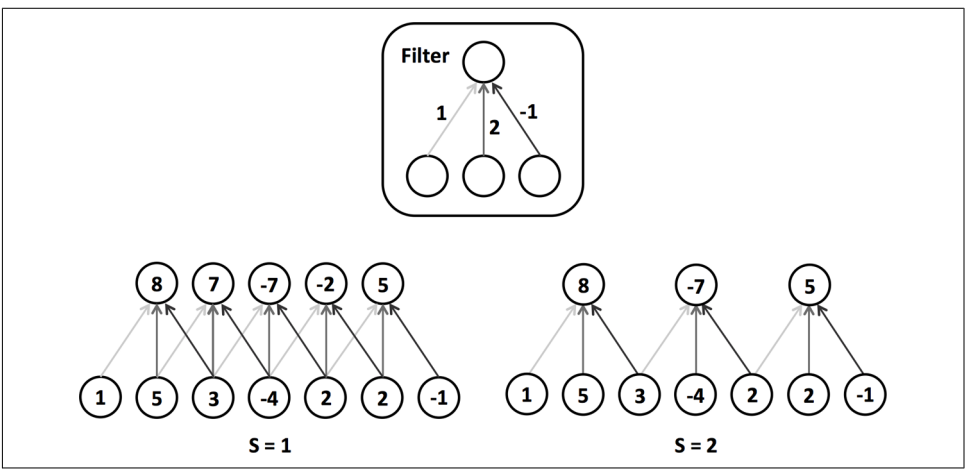


Figure 5-10. An illustration of a filter's stride hyperparameter

This results in an output volume with the following characteristics:

- Its function f, which is applied to the incoming logit of each neuron in the output volume to determine its final value
- Its *width* $w_{out} = \left\lceil \frac{w_{in} - e + 2p}{s} \right\rceil + 1$

- Its *height* $h_{out} = \left\lceil \frac{h_{in} - e + 2p}{s} \right\rceil + 1$
- Its *depth* $d_{out} = k$

The m^{th} "depth slice" of the output volume, where $1 \leq m \leq k$, corresponds to the function f applied to the sum of the m^{th} filter convoluted over the input volume and the bias b^m. Moreover, this means that per filter, we have $d_{in}e^2$ parameters. In total, that means the layer has $kd_{in}e^2$ parameters and k biases. To demonstrate this in action, we provide an example of a convolutional layer in Figure 5-11 and Figure 5-12 with a 5 x 5 x 3 input volume with zero padding $p = 1$. We'll use two 3 x 3 x 3 filters (spatial extent) with a stride $s = 2$. We'll use a linear function to produce the output volume, which will be of size 3 x 3 x 2.

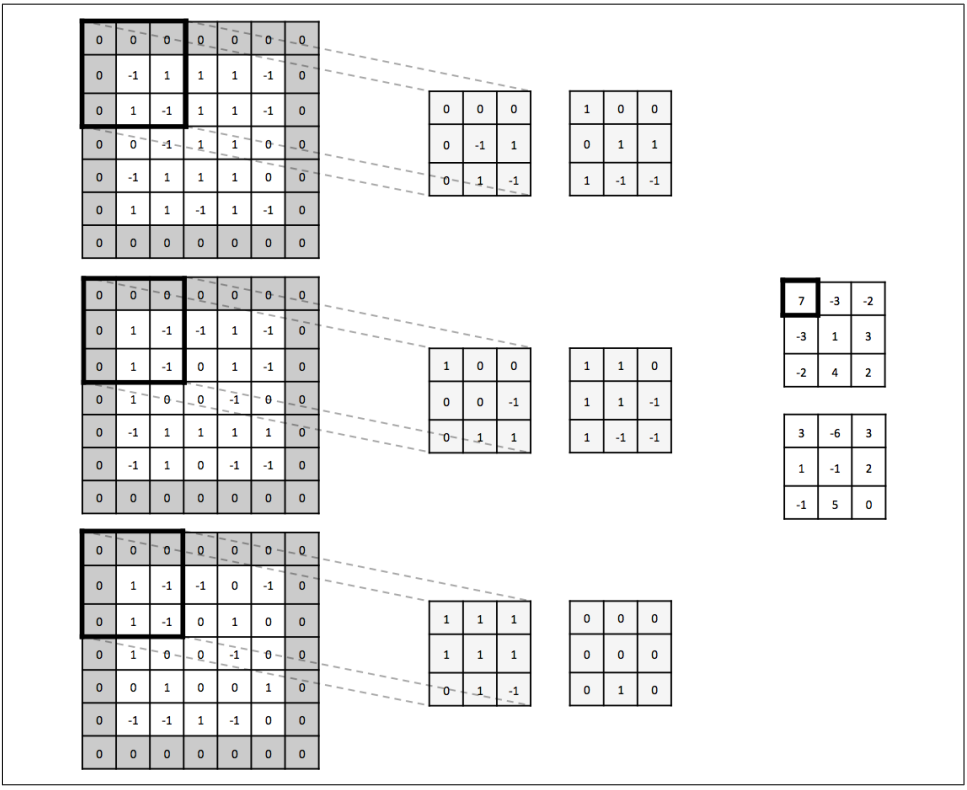


Figure 5-11. This is a convolutional layer with an input volume that has width 5, height 5, depth 3, and zero padding 1. There are 2 filters, with spatial extent 3 and applied with a stride of 2. It results in an output volume with width 3, height 3, and depth 2. We apply the first convolutional filter to the upper-leftmost 3 x 3 piece of the input volume to generate the upper-leftmost entry of the first depth slice.

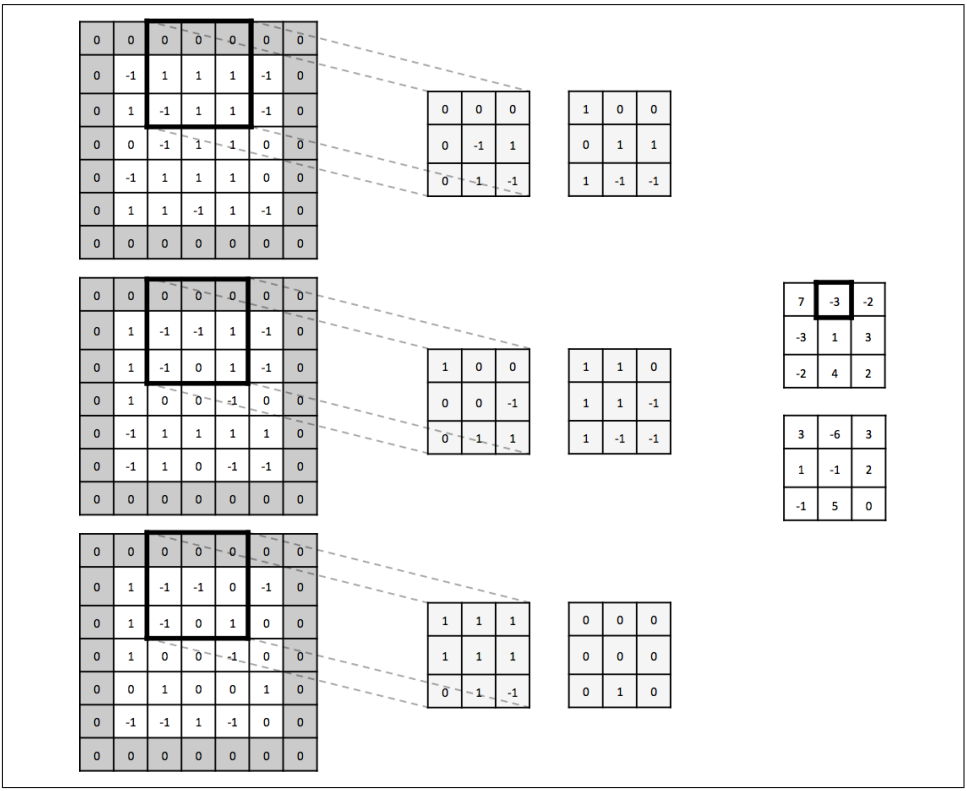


Figure 5-12. Using the same setup as Figure 5-11, we generate the next value in the first depth slice of the output volume.

Generally, it's wise to keep filter sizes small (size 3 x 3 or 5 x 5). Less commonly, larger sizes are used (7 x 7) but only in the first convolutional layer. Having more small filters is an easy way to achieve high representational power while also incurring a smaller number of parameters. It's also suggested to use a stride of 1 to capture all useful information in the feature maps, and a zero padding that keeps the output volume's height and width equivalent to the input volume's height and width.

TensorFlow provides us with a convenient operation to easily perform a convolution on a minibatch of input volumes (note that we must apply our choice of function f ourselves and it is not performed by the operation itself):[9]

```
tf.nn.conv2d(input, filter, strides, padding,
  use_cudnn_on_gpu=True,
             name=None)
```

9 *https://www.tensorflow.org/api_docs/python/tf/nn/conv2d*

Here, `input` is a four-dimensional tensor of size $N \times h_{in} \times w_{in} \times d_{in}$, where N is the number of examples in our minibatch. The `filter` argument is also a four-dimensional tensor representing all of the filters applied in the convolution. It is of size $e \times e \times d_{in} \times k$. The resulting tensor emitted by this operation has the same structure as `input`. Setting the `padding` argument to `"SAME"` also selects the zero padding so that height and width are preserved by the convolutional layer.

Max Pooling

To aggressively reduce dimensionality of feature maps and sharpen the located features, we sometimes insert a *max pooling* layer after a convolutional layer.[10] The essential idea behind max pooling is to break up each feature map into equally sized tiles. Then we create a condensed feature map. Specifically, we create a cell for each tile, compute the maximum value in the tile, and propagate this maximum value into the corresponding cell of the condensed feature map. This process is illustrated in Figure 5-13.

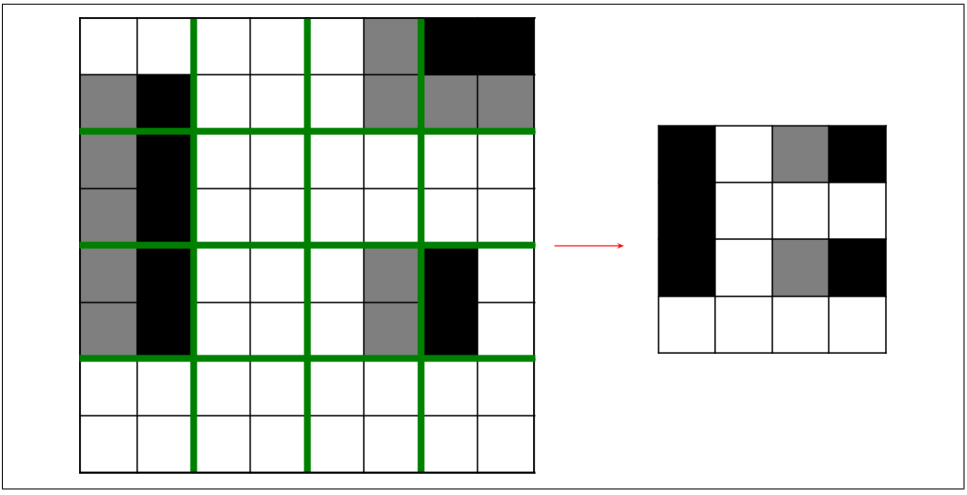


Figure 5-13. An illustration of how max pooling significantly reduces parameters as we move up the network

More rigorously, we can describe a pooling layer with two parameters:

- Its spatial extent e
- Its stride s

10 *https://www.tensorflow.org/api_docs/python/tf/nn/max_pool*

It's important to note that only two major variations of the pooling layer are used. The first is the nonoverlapping pooling layer with $e = 2, s = 2$. The second is the overlapping pooling layer with $e = 3, s = 2$. The resulting dimensions of each feature map are as follows:

- Its width $w_{out} = \left\lceil \frac{w_{in} - e}{s} \right\rceil + 1$
- Its height $h_{out} = \left\lceil \frac{h_{in} - e}{s} \right\rceil + 1$

One interesting property of max pooling is that it is *locally invariant*. This means that even if the inputs shift around a little bit, the output of the max pooling layer stays constant. This has important implications for visual algorithms. Local invariance is a very useful property if we care more about whether some feature is present than exactly where it is. However, enforcing large amounts of local invariance can destroy our network's ability to carry important information. As a result, we usually keep the spatial extent of our pooling layers quite small.

Some recent work along this line has come out of the University of Warwick from Graham[11], who proposes a concept called *fractional max pooling*. In fractional max pooling, a pseudorandom number generator is used to generate tilings with noninteger lengths for pooling. Here, fractional max pooling functions as a strong regularizer, helping prevent overfitting in convolutional networks.

Full Architectural Description of Convolution Networks

Now that we've described the building blocks of convolutional networks, we start putting them together. Figure 5-14 depicts several architectures that might be of practical use.

11 Graham, Benjamin. "Fractional Max-Pooling." *arXiv Preprint arXiv*:1412.6071 (2014).

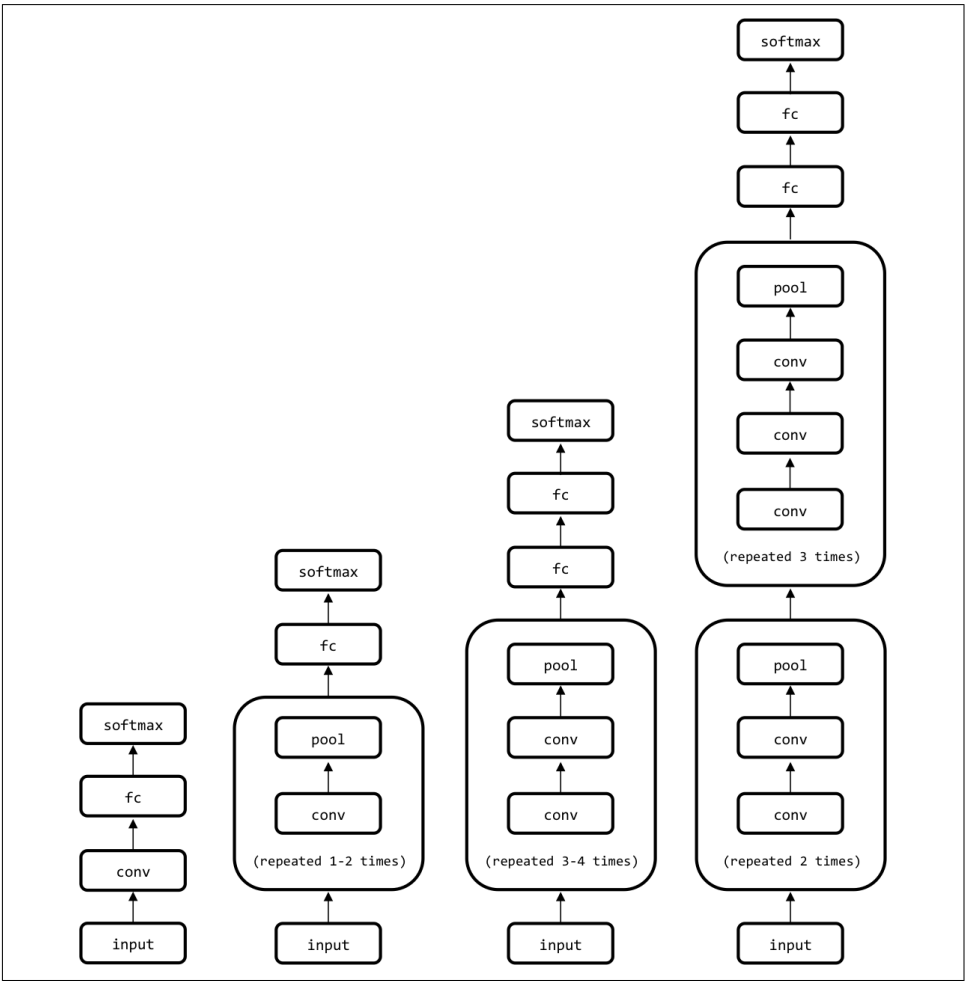


Figure 5-14. Various convolutional network architectures of various complexities. The architecture of VGGNet, a deep convolutional network built for ImageNet, is shown in the rightmost network.

One theme we notice as we build deeper networks is that we reduce the number of pooling layers and instead stack multiple convolutional layers in tandem. This is generally helpful because pooling operations are inherently destructive. Stacking several convolutional layers before each pooling layer allows us to achieve richer representations.

As a practical note, deep convolutional networks can take up a significant amount of space, and most casual practitioners are usually bottlenecked by the memory capacity on their GPU. The VGGNet architecture, for example, takes approximately 90 MB of memory on the forward pass per image and more than 180 MB of memory on the

backward pass to update the parameters.[12] Many deep networks make a compromise by using strides and spatial extents in the first convolutional layer that reduce the amount of information that needs to propagated up the network.

Closing the Loop on MNIST with Convolutional Networks

Now that we have a better understanding of how to build networks that effectively analyze images, we'll revisit the MNIST challenge we've tackled over the past several chapters. Here, we'll use a convolutional network to learn how to recognize handwritten digits. Our feed-forward network was able to achieve a 98.2% accuracy. Our goal will be to push the envelope on this result.

To tackle this challenge, we'll build a convolutional network with a pretty standard architecture (modeled after the second network in Figure 5-14): two pooling and two convolutional interleaved, followed by a fully connected layer (with dropout, $p = 0.5$) and a terminal softmax. To make building the network easy, we write a couple of helper methods in addition to our `layer` generator from the feed-forward network:

```
def conv2d(input, weight_shape, bias_shape):
    in = weight_shape[0] * weight_shape[1] * weight_shape[2]
    weight_init = tf.random_normal_initializer(stddev=
                                               (2.0/in)**0.5)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    bias_init = tf.constant_initializer(value=0)
    b = tf.get_variable("b", bias_shape, initializer=bias_init)
    conv_out = tf.nn.conv2d(input, W, strides=[1, 1, 1, 1],
                            padding='SAME')
    return tf.nn.relu(tf.nn.bias_add(conv_out, b))

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

The first helper method generates a convolutional layer with a particular shape. We set the stride to be to be 1 and the padding to keep the width and height constant between input and output tensors. We also initialize the weights using the same heuristic we used in the feed-forward network. In this case, however, the number of incoming weights into a neuron spans the filter's height and width and the input tensor's depth.

12 Simonyan, Karen, and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *arXiv Preprint arXiv*:1409.1556 (2014).

The second helper method generates a max pooling layer with non-overlapping windows of size k. The default, as recommended, is k=2, and we'll use this default in our MNIST convolutional network.

With these helper methods, we can now build a new inference constructor:

```
def inference(x, keep_prob):

    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 1, 32], [32])
        pool_1 = max_pool(conv_1)

    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 32, 64], [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope("fc"):
        pool_2_flat = tf.reshape(pool_2, [-1, 7 * 7 * 64])
        fc_1 = layer(pool_2_flat, [7*7*64, 1024], [1024])

        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)

    with tf.variable_scope("output"):
        output = layer(fc_1_drop, [1024, 10], [10])

    return output
```

The code here is quite easy to follow. We first take the flattened versions of the input pixel values and reshape them into a tensor of the $N \times 28 \times 28 \times 1$, where N is the number of examples in a minibatch, 28 is the width and height of each image, and 1 is the depth (because the images are black and white; if the images were in RGB color, the depth would instead be 3 to represent each color map). We then build a convolutional layer with 32 filters that have spatial extent 5. This results in taking an input volume of depth 1 and emitting a output tensor of depth 32. This is then passed through a max pooling layer which compresses the information. We then build a second convolutional layer with 64 filters, again with spatial extent 5, taking an input tensor of depth 32 and emitting an output tensor of depth 64. This, again, is passed through a max pooling layer to compress information.

We then prepare to pass the output of the max pooling layer into a fully connected layer. To do this, we flatten the tensor. We can do this by computing the full size of each "subtensor" in the minibatch. We have 64 filters, which corresponds to the depth of 64. We now have to determine the height and width after passing through two max pooling layers. Using the formulas we found in the previous section, it's easy to confirm that each feature map has a height and width of 7. Confirming this is left as an exercise for the reader.

After the reshaping operation, we use a fully connected layer to compress the flattened representation into a hidden state of size 1,024. We use a dropout probability in this layer of 0.5 during training and 1 during model evaluation (standard procedure for employing dropout). Finally, we send this hidden state into a softmax output layer with 10 bins (the softmax is, as usual, performed in the `loss` constructor for better performance).

Finally, we train our network using the Adam optimizer. After several epochs over the dataset, we achieve an accuracy of 99.4%, which isn't state of the art (approximately 99.7 to 99.8%), but is very respectable.

Image Preprocessing Pipelines Enable More Robust Models

So far we've been dealing with rather tame datasets. Why is MNIST a tame dataset? Well, fundamentally, MNIST has already been preprocessed so that all the images in the dataset resemble each other. The handwritten digits are perfectly cropped in just the same way; there are no color aberrations because MNIST is black and white; and so on. Natural images, however, are an entirely different beast.

Natural images are messy, and as a result, there are a number of preprocessing operations that we can utilize in order to make training slightly easier. The first technique that is supported out of the box in TensorFlow is approximate per-image whitening. The basic idea behind whitening is to zero-center every pixel in an image by subtracting out the mean and normalizing to unit 1 variance. This helps us correct for potential differences in dynamic range between images. In TensorFlow, we can achieve this using:

```
tf.image.per_image_whitening(image)
```

We also can expand our dataset artificially by randomly cropping the image, flipping the image, modifying saturation, modifying brightness, etc:

```
tf.random_crop(value, size, seed=None, name=None)
tf.image.random_flip_up_down(image, seed=None)
tf.image.random_flip_left_right(image, seed=None)
tf.image.transpose_image(image)
tf.image.random_brightness(image, max_delta, seed=None)
tf.image.random_contrast(image, lower, upper, seed=None)
tf.image.random_saturation(image, lower, upper, seed=None)
tf.image.random_hue(image, max_delta, seed=None)
```

Applying these transformations helps us build networks that are robust to the different kinds of variations that are present in natural images, and make predictions with high fidelity in spite of potential distortions.

Accelerating Training with Batch Normalization

In 2015, researchers from Google devised an exciting way to even further accelerate the training of feed-forward and convolutional neural networks using a technique called *batch normalization*.[13] We can think of the intuition behind batch normalization like a tower of blocks, as shown in Figure 5-15.

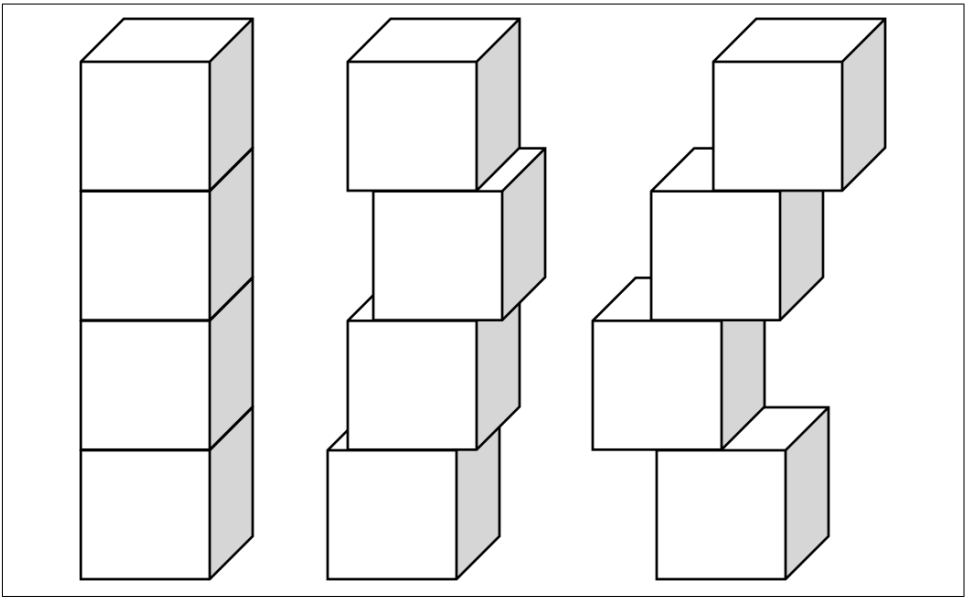


Figure 5-15. When blocks in a tower become shifted too drastically so that they no longer align, the structure can become very unstable

When a tower of blocks is stacked together neatly, the structure is stable. However, if we randomly shift the blocks, we could force the tower into configurations that are increasingly unstable. Eventually the tower falls apart.

A similar phenomenon can happen during the training of neural networks. Imagine a two-layer neural network. In the process of training the weights of the network, the output distribution of the neurons in the bottom layer begins to shift. The result of the changing distribution of outputs from the bottom layer means that the top layer not only has to learn how to make the appropriate predictions, but it also needs to somehow modify itself to accommodate the shifts in incoming distribution. This significantly slows down training, and the magnitude of the problem compounds the more layers we have in our networks.

13 S. Ioffe, C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *arXiv Preprint arXiv*:1502.03167. 2015.

Normalization of image inputs helps out the training process by making it more robust to variations. Batch normalization takes this a step further by normalizing inputs to every layer in our neural network. Specifically, we modify the architecture of our network to include operations that:

1. Grab the vector of logits incoming to a layer before they pass through the nonlinearity
2. Normalize each component of the vector of logits across all examples of the minibatch by subtracting the mean and dividing by the standard deviation (we keep track of the moments using an exponentially weighted moving average)
3. Given normalized inputs $\hat{\mathbf{x}}$, use an affine transform to restore representational power with two vectors of (trainable) parameters: $\gamma\hat{\mathbf{x}} + \beta$

Expressed in TensorFlow, batch normalization can be expressed as follows for a convolutional layer:

```
def conv_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0,
                                        dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0,
                                         dtype=tf.float32)

    beta = tf.get_variable("beta", [n_out],
                           initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out],
                            initializer=gamma_init)

    batch_mean, batch_var = tf.nn.moments(x, [0,1,2],
                                          name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean),
                        ema.average(batch_var)
    def mean_var_with_update():
        with tf.control_dependencies([ema_apply_op]):
            return tf.identity(batch_mean),
                   tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train,
        mean_var_with_update,
        lambda: (ema_mean, ema_var))

    normed = tf.nn.batch_norm_with_global_normalization(x,
             mean, var, beta, gamma, 1e-3, True)
    return normed
```

We can also express batch normalization for nonconvolutional feedforward layers, with a slight modification to how the moments are calculated, and a reshaping option for compatibility with `tf.nn.batch_norm_with_global_normalization`:

```
def layer_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0,
                                        dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0,
                                         dtype=tf.float32)

    beta = tf.get_variable("beta", [n_out],
                           initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out],
                            initializer=gamma_init)

    batch_mean, batch_var = tf.nn.moments(x, [0],
                                          name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean),
                        ema.average(batch_var)
    def mean_var_with_update():
        with tf.control_dependencies([ema_apply_op]):
            return tf.identity(batch_mean),
                   tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train,
        mean_var_with_update,
        lambda: (ema_mean, ema_var))

    x_r = tf.reshape(x, [-1, 1, 1, n_out])
    normed = tf.nn.batch_norm_with_global_normalization(x_r,
             mean, var, beta, gamma, 1e-3, True)
    return tf.reshape(normed, [-1, n_out])
```

In addition to speeding up training by preventing significant shifts in the distribution of inputs to each layer, batch normalization also allows us to significantly increase the learning rate. Moreover, batch normalization acts as a regularizer and removes the need for dropout and (when used) L2 regularization. Although we don't leverage it here, the authors also claim that batch regularization largely removes the need for photometric distortions, and we can expose the network to more "real" images during the training process.

Now that we've developed an enhanced toolkit for analyzing natural images with convolutional networks, we'll now build a classifier for tackling the CIFAR-10 challenge.

Building a Convolutional Network for CIFAR-10

The CIFAR-10 challenge consists of 32 x 32 color images that belong to one of 10 possible classes.[14] This is a surprisingly hard challenge because it can be difficult for even a human to figure out what is in a picture. An example is shown in Figure 5-16.

Figure 5-16. A dog from the CIFAR-100 dataset

In this section, we'll build networks both with and without batch normalization as a basis of comparison. We increase the learning rate by 10-fold for the batch normalization network to take full advantage of its benefits. We'll only display code for the batch normalization network here because building the vanilla convolutional network is very similar.

We distort random 24 × 24 crops of the input images to feed into our network for training. We use the example code provided by Google to do this. We'll jump right

14 Krizhevsky, Alex, and Geoffrey Hinton. "Learning Multiple Layers of Features from Tiny Images." (2009).

into the network architecture. To start, let's take a look at how we integrate batch normalization into the convolutional and fully connected layers. As expected, batch normalization happens to the logits before they're fed into a nonlinearity:

```
def conv2d(input, weight_shape, bias_shape, phase_train,
           visualize=False):
    incoming = weight_shape[0] * weight_shape[1]
                               * weight_shape[2]
    weight_init = tf.random_normal_initializer(stddev=
                  (2.0/incoming)**0.5)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    if visualize:
        filter_summary(W, weight_shape)
    bias_init = tf.constant_initializer(value=0)
    b = tf.get_variable("b", bias_shape, initializer=bias_init)
    logits = tf.nn.bias_add(tf.nn.conv2d(input, W,
             strides=[1, 1, 1, 1], padding='SAME'), b)
    return tf.nn.relu(conv_batch_norm(logits, weight_shape[3],
                                      phase_train))

def layer(input, weight_shape, bias_shape, phase_train):
    weight_init = tf.random_normal_initializer(stddev=
                  (2.0/weight_shape[0])**0.5)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    b = tf.get_variable("b", bias_shape,
                        initializer=bias_init)
    logits = tf.matmul(input, W) + b
    return tf.nn.relu(layer_batch_norm(logits, weight_shape[1],
                      phase_train))
```

The rest of the architecture is straightforward. We use two convolutional layers (each followed by a max pooling layer). There are then two fully connected layers followed by a softmax. Dropout is included for reference, but in the batch normalization version, `keep_prob=1` during training:

```
def inference(x, keep_prob, phase_train):

    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 3, 64], [64], phase_train,
                        visualize=True)
        pool_1 = max_pool(conv_1)

    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 64, 64], [64],
                        phase_train)
        pool_2 = max_pool(conv_2)

    with tf.variable_scope("fc_1"):
```

```
        dim = 1
        for d in pool_2.get_shape()[1:].as_list():
            dim *= d

        pool_2_flat = tf.reshape(pool_2, [-1, dim])
        fc_1 = layer(pool_2_flat, [dim, 384], [384],
                     phase_train)

        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)

    with tf.variable_scope("fc_2"):

        fc_2 = layer(fc_1_drop, [384, 192], [192], phase_train)

        # apply dropout
        fc_2_drop = tf.nn.dropout(fc_2, keep_prob)

    with tf.variable_scope("output"):
        output = layer(fc_2_drop, [192, 10], [10], phase_train)

    return output
```

Finally, we use the Adam optimizer to train our convolutional networks. After some amount of time training, our networks are able to achieve an impressive 92.3% accuracy on the CIFAR-10 task without batch normalization and 96.7% accuracy with batch normalization. This result actually matches (and potentially exceeds) current state-of-the-art research on this task! In the next section, we'll take a closer look at learning and visualize how our networks perform.

Visualizing Learning in Convolutional Networks

On a high level, the simplest thing that we can do to visualize training is plot the cost function and validation errors over time as training progresses. We can clearly demonstrate the benefits of batch normalization by comparing the rates of convergence between our two networks. Plots taken in the middle of the training process are shown in Figure 5-17.

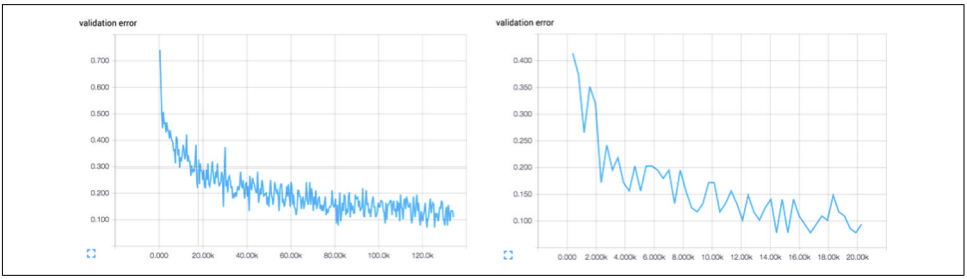


Figure 5-17. Training a convolutional network without batch normalization (left) versus with batch normalization (right). Batch normalization vastly accelerates the training process.

Without batch normalization, cracking the 90% accuracy threshold requires over 80,000 minibatches. On the other hand, with batch normalization, crossing the same threshold only requires slightly over 14,000 minibatches.

We can also inspect the filters that our convolutional network learns in order to understand what the network finds important to its classification decisions. Convolutional layers learn hierarchical representations, and so we'd hope that the first convolutional layer learns basic features (edges, simple curves, etc.), and the second convolutional layer will learn more complex features. Unfortunately, the second convolutional layer is difficult to interpret even if we decided to visualize it, so we only include the first layer filters in Figure 5-18.

Figure 5-18. A subset of the learned filters in the first convolutional layer of our network

We can make out a number of interesting features in our filters: vertical, horizontal, and diagonal edges, in addition to small dots or splotches of one color surrounded by another. We can be confident that our network is learning relevant features because the filters are not just noise.

We can also try to visualize how our network has learned to cluster various kinds of images pictorially. To illustrate this, we take a large network that has been trained on the ImageNet challenge and then grab the hidden state of the fully connected layer just before the softmax for each image. We then take this high-dimensional representation for each image and use an algorithm known as *t-Distributed Stochastic Neighbor Embedding*, or *t-SNE*, to compress it to a two-dimensional representation that we can visualize.[15] We don't cover the details of t-SNE here, but there are a number of

15 Maaten, Laurens van der, and Geoffrey Hinton. "Visualizing Data using t-SNE." *Journal of Machine Learning Research* 9.Nov (2008): 2579-2605.

publicly available software tools that will do it for us, including the script. We visualize the embeddings in Figure 5-19, and the results are quite spectacular.

Figure 5-19. The t-SNE embedding (center) surrounded by zoomed-in subsegments of the embedding (periphery). Image credit: Andrej Karpathy.[16]

At first, on a high level, it seems that images that are similarly colored are closer together. This is interesting, but what's even more striking is when we zoom into parts of the visualization, we realize that it's more than just color. We realize that all pictures of boats are in one place, all pictures of humans are in another place, and all pictures of butterflies are in yet another location in the visualization. Quite clearly, convolutional networks have spectacular learning capabilities.

16 *http://cs.stanford.edu/people/karpathy/cnnembed/*

Leveraging Convolutional Filters to Replicate Artistic Styles

Over the past couple of years, we've also developed algorithms that leverage convolutional networks in much more creative ways. One of these algorithms is called *neural style*.[17] The goal of neural style is to be able to take an arbitrary photograph and rerender it as if it were painted in the style of a famous artist. This seems like a daunting task, and it's not exactly clear how we might approach this problem if we didn't have a convolutional network. However, it turns out that clever manipulation of convolutional filters can produce spectacular results on this problem.

Let's take a pre-trained convolutional network. There are three images that we're dealing with. The first two are the source of content $\boldsymbol{p}$ and the source of style $\boldsymbol{a}$. The third image is the generated image $\boldsymbol{x}$. Our goal will be to derive an error function that we can backpropagate that, when minimized, will perfectly combine the content of the desired photograph and the style of the desired artwork.

We start with content first. If a layer in the network has k_l filters, then it produces a total of k_l feature maps. Let's call the size of each feature map m_l, the height times the width of the feature map. This means that the activations in all the feature maps of this layer can be stored in a matrix $\boldsymbol{F}^{(l)}$ of size $k_l \times m_l$. We can also represent all the activations of the photograph in a matrix $\boldsymbol{P}^{(l)}$ and all the activations of the generated image in the matrix $\boldsymbol{X}^{(l)}$. We use the `relu4_2` of the original VGGNet:

$$E_{\text{content}}(\boldsymbol{p}, \boldsymbol{x}) = \sum_{\text{ij}}(\boldsymbol{P}_{\text{ij}}^{(l)} - \boldsymbol{X}_{\text{ij}}^{(l)})^2$$

Now we can try tackling style. To do this we construct a matrix known as the *Gram matrix*, which represents correlations between feature maps in a given layer. The correlations represent the texture and feel that is common among all features, irrespective of which features we're looking at. Constructing the Gram matrix, which is of size $k_l \times k_l$, for a given image is done as follows:

$$\mathbf{G}^{(l)}_{\text{ij}} = \sum_{\text{c}=0}^{\text{ml}} \mathbf{F}^{(l)}_{ic} \mathbf{F}^{(l)}_{jc}$$

We can compute the Gram matrices for both the artwork in matrix $\boldsymbol{A}^{(l)}$ and the generated image in $\boldsymbol{G}^{(l)}$. We can then represent the error function as:

$$E_{style}(\mathbf{a}, \mathbf{x}) = \frac{1}{4k_l^2 m_l^2} \sum_{l=1}^{L} \sum_{ij} \frac{1}{L} \left(A_{ij}^{(l)} - G_{ij}^{(l)} \right)^2$$

Here, we weight each squared difference equally (dividing by the number of layers we want to include in our style reconstruction). Specifically, we use the `relu1_1`,

17 Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style." *arXiv Preprint arXiv*:1508.06576 (2015).

`relu2_1`, `relu3_1`, `relu4_1`, and `relu5_1` layers of the original VGGNet. We omit full a discussion of the TensorFlow code (*http://bit.ly/2qAODnp*) for brevity, but the results, as shown in Figure 5-20, are again quite spectacular. We mix a photograph of the iconic MIT dome and Leonid Afremov's *Rain Princess*.

Figure 5-20. The result of mixing the Rain Princess with a photograph of the MIT Dome. Image credit: Anish Athalye.

Learning Convolutional Filters for Other Problem Domains

Although our examples in this chapter focus on image recognition, there are several other problem domains in which convolutional networks are useful. A natural extension of image analysis is video analysis. In fact, using five-dimensional tensors (including time as a dimension) and applying three-dimensional convolutions is an easy way to extend the convolutional paradigm to video.[18] Convolutional filters have

18 Karpathy, Andrej, et al. "Large-scale Video Classification with Convolutional Neural Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014.

also been successfully used to analyze audiograms.[19] In these applications, a convolutional network slides over an audiogram input to predict phonemes on the other side.

Less intuitively, convolutional networks have also found some use in natural language processing. We'll see some examples of this in later chapters. More exotic uses of convolutional networks include teach algorithms to play board games, and analyzing biological molecules for drug discovery. We'll also discuss both of these examples in later chapters of this book.

Summary

In this chapter, we learned how to build neural networks that analyze images. We developed the concept of a convolution, and leveraged this idea to create tractable networks that can analyze both simple and more complex natural images. We built several of these convolutional networks in TensorFlow, and leveraged various image processing pipelines and batch normalization to make training our networks faster and more robust. Finally, we visualized the learning of convolutional networks and explored other interesting applications of the technology.

Images were easy to analyze because we were able to come up with effective ways to represent them as tensors. In other situations (e.g., natural language), it's less clear how one might represent our input data as tensors. To tackle this problem as a stepping stone to new deep learning models, we'll develop some key concepts in vector embeddings and representation learning in the next chapter.

19 Abdel-Hamid, Ossama, et al. "Applying Convolutional Neural Networks concepts to hybrid NN-HMM model for speech recognition." IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto, 2012, pp. 4277-4280.

CHAPTER 6

Embedding and Representation Learning

Learning Lower-Dimensional Representations

In the previous chapter, we motivated the convolutional architecture using a simple argument. The larger our input vector, the larger our model. Large models with lots of parameters are expressive, but they're also increasingly data hungry. This means that without sufficiently large volumes of training data, we will likely overfit. Convolutional architectures help us cope with the curse of dimensionality by reducing the number of parameters in our models without necessarily diminishing expressiveness.

Regardless, convolutional networks still require large amounts of labeled training data. And for many problems, labeled data is scarce and expensive to generate. Our goal in this chapter will be to develop effective learning models in situations where labeled data is scarce but wild, unlabeled data is plentiful. We'll approach this problem by learning *embeddings*, or low-dimensional representations, in an unsupervised fashion. Because these unsupervised models allow us to offload all of the heavy lifting of automated feature selection, we can use the generated embeddings to solve learning problems using smaller models that require less data. This process is summarized in Figure 6-1.

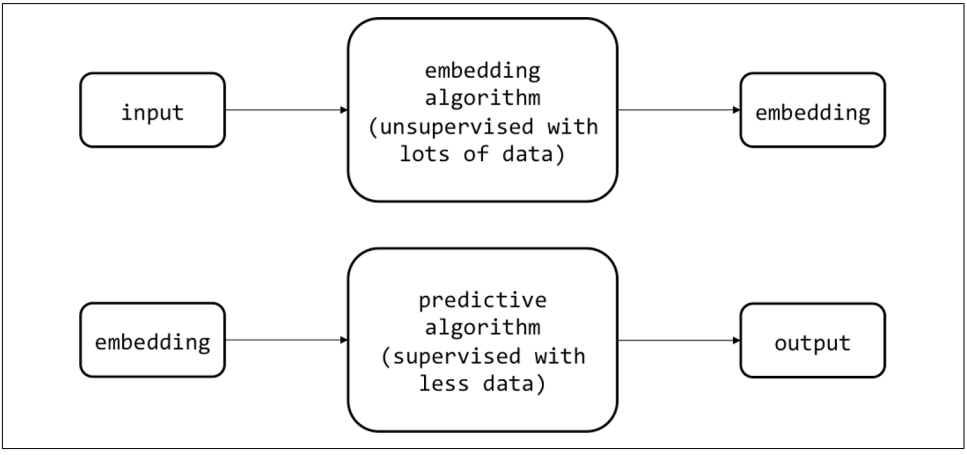


Figure 6-1. Using embeddings to automate feature selection in the face of scarce labeled data

In the process of developing algorithms that learn good embeddings, we'll also explore other applications of learning lower-dimensional representations, such as visualization and semantic hashing. We'll start by considering situations where all of the important information is already contained within the original input vector itself. In this case, learning embeddings is equivalent to developing an effective compression algorithm.

In the next section, we'll introduce *principal component analysis (PCA)*, a classic method for dimensionality reduction. In subsequent sections, we'll explore more powerful neural methods for learning compressive embeddings.

Principal Component Analysis

The basic concept behind PCA is that we'd like to find a set of axes that communicates the most information about our dataset. More specifically, if we have d-dimensional data, we'd like to find a new set of $m < d$ dimensions that conserves as much valuable information from the original dataset. For simplicity, let's choose $d = 2, m = 1$. Assuming that variance corresponds to information, we can perform this transformation through an iterative process. First we find a unit vector along which the dataset has maximum variance. Because this direction contains the most information, we select this direction as our first axis. Then from the set of vectors orthogonal to this first choice, we pick a new unit vector along which the dataset has maximum variance. This is our second axis. We continue this process until we have found a total of d new vectors that represent new axes. We project our data onto this new set of axes. We then decide a good value for m and toss out all but the

first m axes (the principal components, which store the most information). The result is shown in Figure 6-2.

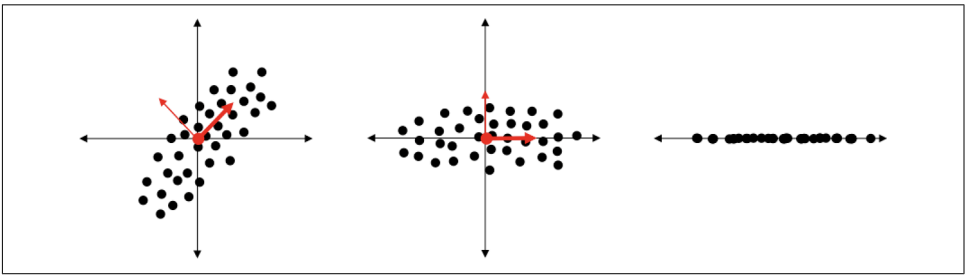


Figure 6-2. An illustration of PCA for dimensionality reduction to capture the dimension with the most information (as proxied by variance)

For the mathematically initiated, we can view this operation as a project onto the vector space spanned by the top m eigenvectors of the dataset's covariance matrix (within constant scaling). Let us represent the dataset as a matrix $\mathbf{X}$ with dimensions $n \times d$ (i.e., n inputs of d dimensions). We'd like to create an embedding matrix $\mathbf{T}$ with dimensions $n \times m$. We can compute the matrix using the relationship $\mathbf{T} = \mathbf{X}$, where each column of $\mathbf{W}$ corresponds to an eigenvector of the matrix $\mathbf{X}^T\mathbf{X}$.

While PCA has been used for decades for dimensionality reduction, it spectacularly fails to capture important relationships that are piecewise linear or nonlinear. Take, for instance, the example illustrated in Figure 6-3.

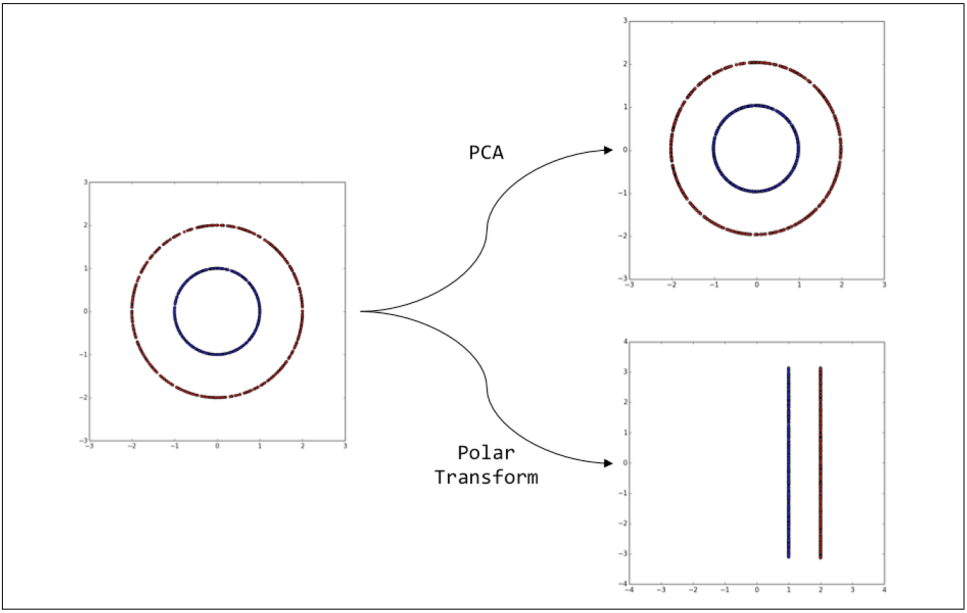


Figure 6-3. A situation in which PCA fails to optimally transform the data for dimensionality reduction

The example shows data points selected at random from two concentric circles. We hope that PCA will transform this dataset so that we can pick a single new axis that allows us to easily separate the red and blue dots. Unfortunately for us, there is no linear direction that contains more information here than another (we have equal variance in all directions). Instead, as a human being, we notice that information is being encoded in a nonlinear way, in terms of how far points are from the origin. With this information in mind, we notice that the polar transformation (expressing points as their distance from the origin, as the new horizontal axis, and their angle bearing from the original x-axis, as the new vertical axis) does just the trick.

Figure 6-3 highlights the shortcomings of an approach like PCA in capturing important relationships in complex datasets. Because most of the datasets we are likely to encounter in the wild (images, text, etc.) are characterized by nonlinear relationships, we must develop a theory that will perform nonlinear dimensionality reduction. Deep learning practitioners have closed this gap using neural models, which we'll cover in the next section.

Motivating the Autoencoder Architecture

When we talked about feed-forward networks, we discussed how each layer learned progressively more relevant representations of the input. In fact, in Chapter 5, we took the output of the final convolutional layer and used that as a lower-dimensional

representation of the input image. Putting aside the fact that we want to generate these low-dimensional representations in an unsupervised fashion, there are fundamental problems with these approaches in general. Specifically, while the selected layer does contain information from the input, the network has been trained to pay attention to the aspects of the input that are critical to solving the task at hand. As a result, there's a significant amount of information loss with respect to elements of the input that may be important for other classification tasks, but potentially less important than the one immediately at hand.

However, the fundamental intuition here still applies. We define a new network architecture that we call the *autoencoder*. We first take the input and compress it into a low-dimensional vector. This part of the network is called the *encoder* because it is responsible for producing the low-dimensional embedding or *code*. The second part of the network, instead of mapping the embedding to an arbitrary label as we would in a feed-forward network, tries to invert the computation of the first half of the network and reconstruct the original input. This piece is known as the *decoder*. The overall architecture is illustrated in Figure 6-4.

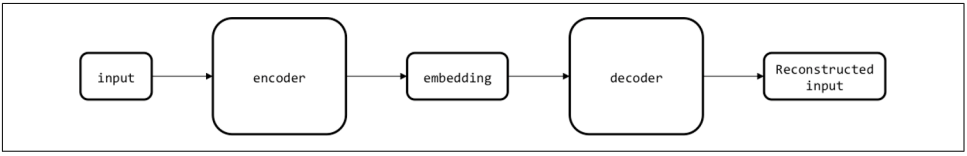


Figure 6-4. The autoencoder architecture attempts to construct a high-dimensional input into a low-dimensional embedding and then uses that low-dimensional embedding to reconstruct the input

To demonstrate the surprising effectiveness of autoencoders, we'll build and visualize the autoencoder architecture in Figure 6-5. Specifically, we will highlight its superior ability to separate MNIST digits as compared to PCA.

Implementing an Autoencoder in TensorFlow

The seminal paper "Reducing the dimensionality of data with neural networks," which describes the autoencoder, was written by Hinton and Salakhutdinov in 2006.[1] Their hypothesis was that the nonlinear complexities afforded by a neural model would allow them to capture structure that linear methods, such as PCA, would miss. To demonstrate this point, they ran an experiment on MNIST using both an autoencoder and PCA to reduce the dataset into two-dimensional data points. In this sec-

1 Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks." *Science* 313.5786 (2006): 504-507.

tion, we will recreate their experimental setup to validate this hypothesis and further explore the architecture and properties of feed-forward autoencoders.

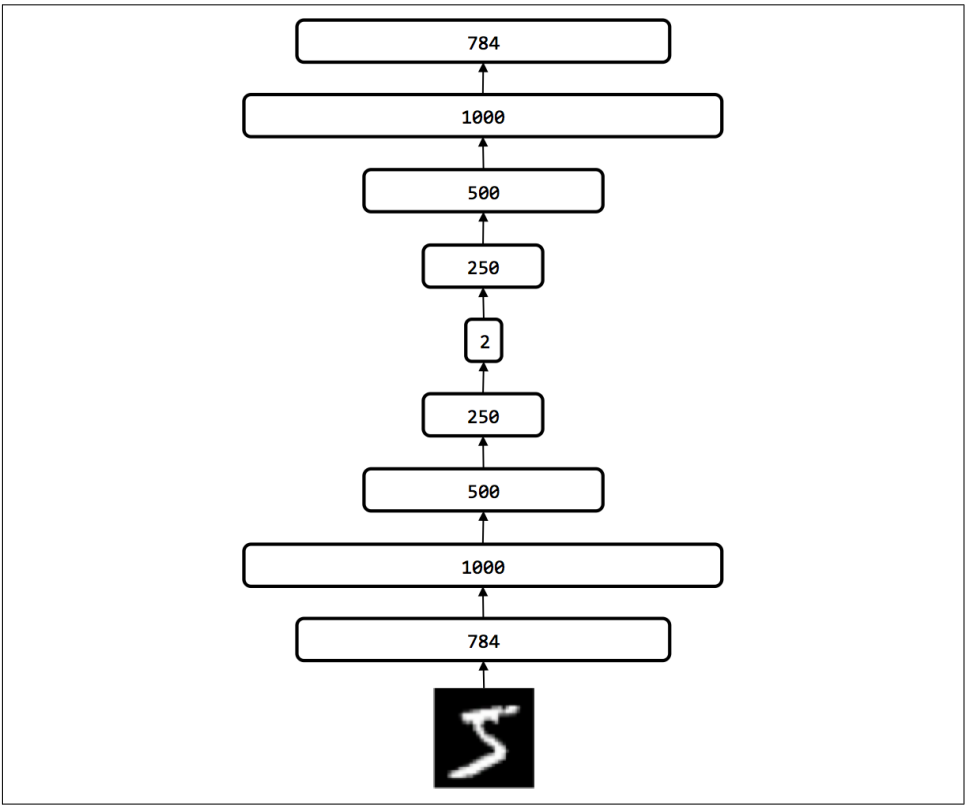


Figure 6-5. The experimental setup for dimensionality reduction of the MNIST dataset employed by Hinton and Salakhutdinov, 2006

The setup shown in Figure 6-5 is built with the same principle, but the two-dimensional embedding is now treated as the input, and the network attempts to reconstruct the original image. Because we are essentially applying an inverse operation, we architect the decoder network so that the autoencoder has the shape of an hourglass. The output of the decoder network is a 784-dimensional vector that can be reconstructed into a 28 × 28 image:

```
def decoder(code, n_code, phase_train):
    with tf.variable_scope("decoder"):
        with tf.variable_scope("hidden_1"):
            hidden_1 = layer(code, [n_code, n_decoder_hidden_1],
                       [n_decoder_hidden_1], phase_train)

        with tf.variable_scope("hidden_2"):
            hidden_2 = layer(hidden_1, [n_decoder_hidden_1,
```

```
                        n_decoder_hidden_2], [n_decoder_hidden_2],
phase_train)

        with tf.variable_scope("hidden_3"):
            hidden_3 = layer(hidden_2, [n_decoder_hidden_2,
                        n_decoder_hidden_3], [n_decoder_hidden_3],
phase_train)

        with tf.variable_scope("output"):
            output = layer(hidden_3, [n_decoder_hidden_3, 784],
                    [784], phase_train)

    return output
```

As a quick note, in order to accelerate training, we'll reuse the batch normalization strategy we employed in Chapter 5. Also, because we'd like to visualize the results, we'll avoid introducing sharp transitions in our neurons. In this example, we'll use sigmoidal neurons instead of our usual ReLU neurons:

```
def layer(input, weight_shape, bias_shape, phase_train):
    weight_init = tf.random_normal_initializer(stddev=
                  (1.0/weight_shape[0])**0.5)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    b = tf.get_variable("b", bias_shape,
                        initializer=bias_init)
    logits = tf.matmul(input, W) + b
    return tf.nn.sigmoid(layer_batch_norm(logits,
                                    weight_shape[1],
                     phase_train))
```

Finally, we need to construct a measure (or objective function) that describes how well our model functions. Specifically, we want to measure how close the reconstruction is to the original image. We can measure this simply by computing the distance between the original 784-dimensional input and the reconstructed 784-dimensional output. More specifically, given an input vector I and a reconstruction O, we'd like to minimize the value of $\| I - O \| = \sqrt{\Sigma_i (I_i - O_i)^2}$, also known as the L2 norm of the difference between the two vectors. We average this function over the whole mini-batch to generate our final objective function. Finally, we'll train the network using the Adam optimizer, logging a scalar summary of the error incurred at every mini-batch using `tf.scalar_summary`. In TensorFlow, we can concisely express the loss and training operations as follows:

```
def loss(output, x):
    with tf.variable_scope("training"):
        l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(output, x)),
   1))
        train_loss = tf.reduce_mean(l2)
        train_summary_op = tf.scalar_summary("train_cost",
  train_loss)
        return train_loss, train_summary_op

def training(cost, global_step):
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001,
                beta1=0.9, beta2=0.999, epsilon=1e-08,
                use_locking=False, name='Adam')
    train_op = optimizer.minimize(cost, global_step=global_step)
    return train_op
```

Finally, we'll need a method to evaluate the generalizability of our model. As usual, we'll use a validation dataset and compute the same L2 norm measurement for model evaluation. In addition, we'll collect image summaries so that we can compare both the input images and the reconstructions:

```
def image_summary(summary_label, tensor):
    tensor_reshaped = tf.reshape(tensor, [-1, 28, 28, 1])
    return tf.image_summary(summary_label, tensor_reshaped)

def evaluate(output, x):
    with tf.variable_scope("validation"):
        in_im_op = image_summary("input_image", x)
        out_im_op = image_summary("output_image", output)
        l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(output, x,
             name="val_diff")), 1))
        val_loss = tf.reduce_mean(l2)
        val_summary_op = tf.scalar_summary("val_cost", val_loss)
        return val_loss, in_im_op, out_im_op, val_summary_op
```

Finally, all that's left to do is build the model out of these subcomponents and train the model. A lot of this code is familiar, but it has a couple of additional bells and whistles that are worth covering. First, we have modified our usual code to accept a command-line parameter for determining the number of neurons in our code layer. For example, running `$ python autoencoder_mnist.py 2` will instantiate a model with two neurons in the code layer. We also reconfigure the model saver to maintain more snapshots of our model. We'll be reloading our most effective model later to compare its performance to PCA, so we'd like to be able to have access to many snapshots. We use summary writers to also capture the image summaries we generate at the end of each epoch:

```
if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='Test various
                                     optimization strategies')
    parser.add_argument('n_code', nargs=1, type=str)
    args = parser.parse_args()
    n_code = args.n_code[0]

    mnist = input_data.read_data_sets("data/", one_hot=True)

    with tf.Graph().as_default():

        with tf.variable_scope("autoencoder_model"):

            x = tf.placeholder("float", [None, 784]) # mnist
                         data image of shape 28*28=784
            phase_train = tf.placeholder(tf.bool)

            code = encoder(x, int(n_code), phase_train)

            output = decoder(code, int(n_code), phase_train)

            cost, train_summary_op = loss(output, x)

            global_step = tf.Variable(0, name='global_step',
           trainable=False)

            train_op = training(cost, global_step)

            eval_op, in_im_op, out_im_op, val_summary_op =
            evaluate(output, x)

            summary_op = tf.merge_all_summaries()

            saver = tf.train.Saver(max_to_keep=200)

            sess = tf.Session()

            train_writer = tf.train.SummaryWriter(
   "mnist_autoencoder_hidden=" + n_code +
                "_logs/",graph=sess.graph)

            val_writer = tf.train.SummaryWriter(
   "mnist_autoencoder_hidden=" + n_code +
                "_logs/", graph=sess.graph)

            init_op = tf.initialize_all_variables()

            sess.run(init_op)

            # Training cycle
            for epoch in range(training_epochs):
```

```
            avg_cost = 0.
            total_batch = int(mnist.train.num_examples/
    batch_size)
            # Loop over all batches
            for i in range(total_batch):
                mbatch_x, mbatch_y =
mnist.train.next_batch(batch_size)
                # Fit training using batch data
                _, new_cost, train_summary = sess.run([
                                train_op, cost,
                                train_summary_op],
                         feed_dict={x: mbatch_x,
                              phase_train: True})
                train_writer.add_summary(train_summary,
    sess.run(global_step))
                # Compute average loss
                avg_cost += new_cost/total_batch
            # Display logs per epoch step
            if epoch % display_step == 0:
                print "Epoch:", '%04d' % (epoch+1),
   "cost =", "{:.9f}".format(avg_cost)

                train_writer.add_summary(train_summary,
                          sess.run(global_step))
                val_images = mnist.validation.images
                validation_loss, in_im, out_im,
       val_summary = sess.run([eval_op, in_im_op,
       out_im_op, val_summary_op],
       feed_dict={x: val_images,
       phase_train: False})
                val_writer.add_summary(in_im, sess.run
                         (global_step))
                val_writer.add_summary(out_im, sess.run
                         (global_step))
                val_writer.add_summary(val_summary, sess.run
                         (global_step))
                print "Validation Loss:", validation_loss

                saver.save(sess,
               "mnist_autoencoder_hidden=" + n_code +
               "_logs/model-checkpoint-"
                 + '%04d' % (epoch+1),
                        global_step=global_step)


        print "Optimization Finished!"


        test_loss = sess.run(eval_op, feed_dict={x:
      mnist.test.images, phase_train: False})
```

```
print "Test Loss:", loss
```

We can visualize the TensorFlow graph, the training and validation costs, and the image summaries using TensorBoard. Simply run the following command:

```
$ tensorboard --logdir ~/path/to/mnist_autoencoder_hidden=2_logs
```

Then navigate your browser to *http://localhost:6006/*. The results of the "Graph" tab are shown in Figure 6-6.

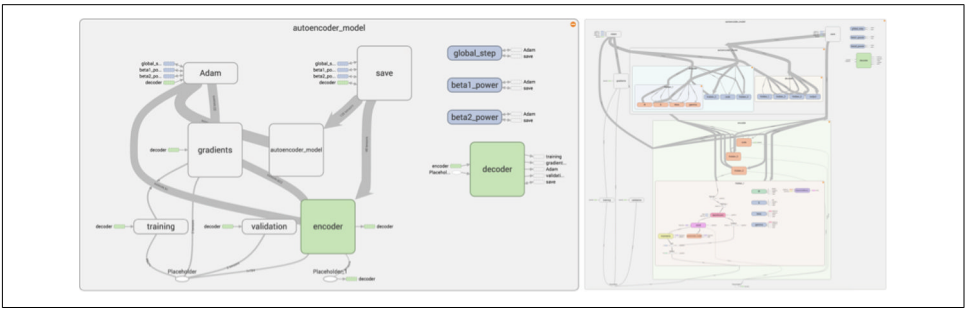


Figure 6-6. TensorFlow allows us to neatly view the high-level components and data flow of our computation graph (left) and also click through to more closely inspect the data flows of individual subcomponents (right)

Thanks to how we've namespaced the components of our TensorFlow graph, our model is nicely organized. We can easily click through the components and delve deeper, tracing how data flows up through the various layers of the encoder and through the decoder, how the optimizer reads the output of our training module, and how gradients in turn affect all of the components of the model.

We also visualize both the training (after each minibatch) and validation costs (after each epoch), closely monitoring the curves for potential overfitting. The TensorBoard visualizations of the costs over the span of training are shown in Figure 6-7. As we would expect for a successful model, both the training and validation curves decrease until they flatten off asymptotically. After approximately 200 epochs, we attain a validation cost of 4.78. While the curves look promising, it's difficult to, upon first glance, understand whether we've reached a plateau at a "good" cost, or whether our model is still doing a poor job of reconstructing the original inputs

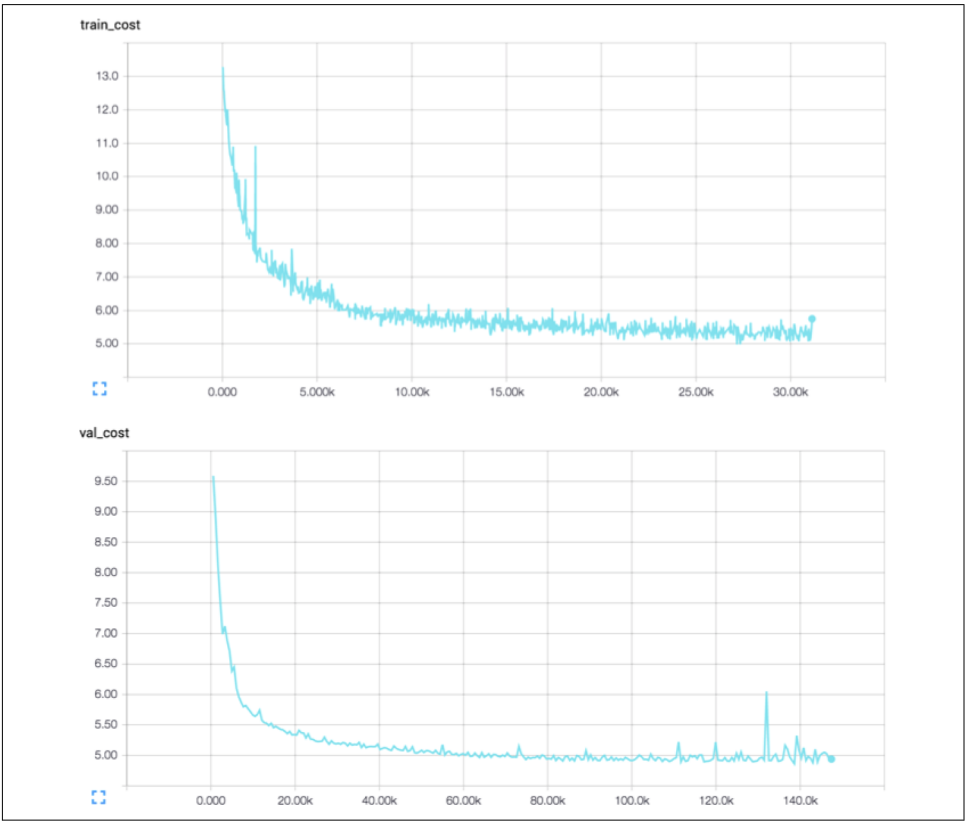


Figure 6-7. The cost incurred on the training set (logged after each minibatch) and on the validation set (logged after each epoch)

To get a sense of what that means, let's explore the MNIST dataset. We pick an arbitrary image of a 1 from the dataset and call it X. In Figure 6-8, we compare the image to all other images in the dataset. Specifically, for each digit class, we compute the average of the L2 costs, comparing X to each instance of the digit class. As a visual aide, we also include the average of all of the instances for each digit class.

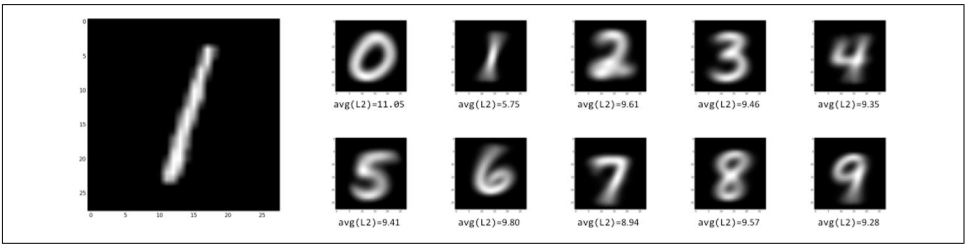


Figure 6-8. The image of the 1 on the left is compared to all of the other digits in the MNIST dataset; each digit class is represented visually with the average of all of its members and labeled with the average of the L2 costs, comparing the 1 on the left with all of the class members

On average, *X* is 5.75 units away from other 1's in MNIST. In terms of L2 distance, the non-1 digits closest to the *X* are the 7's (8.94 units) and the digits farthest are the 0's (11.05 units). Given these measurements, it's quite apparent that with an average cost of 4.78, our autoencoder is producing high-quality reconstructions.

Because we are collecting image summaries, we can confirm this hypothesis directly by inspecting the input images and reconstructions directly. The reconstructions for three randomly chosen samples from the test set are shown in Figure 6-9.

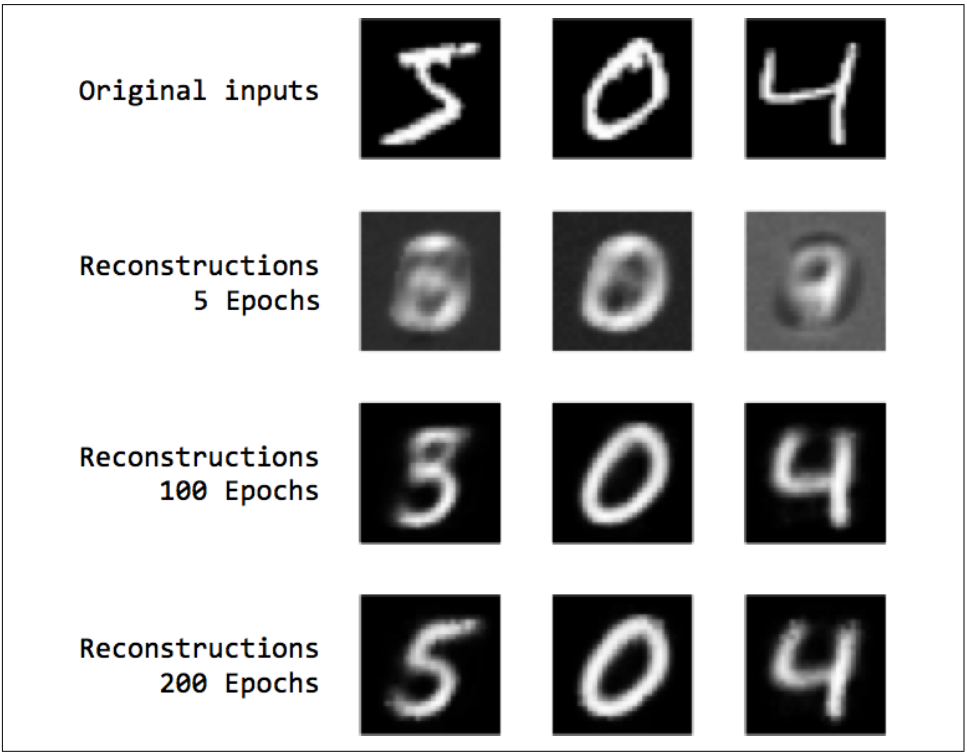


Figure 6-9. A side-by-side comparison of the original inputs (from the validation set) and reconstructions after 5, 100, and 200 epochs of training

After five epochs, we can start to make out some of the critical strokes of the original image that are being picked by the autoencoder, but for the most part, the reconstructions are still hazy mixtures of closely related digits. By 100 epochs, the 0 and 4 are reconstructed with strong strokes, but it looks like the autoencoder is still having trouble differentiating between 5's, 3's, and possibly 8's. However, by 200 epochs, it's clear that even this more difficult ambiguity is clarified, and all of the digits are crisply reconstructed.

Finally, we'll complete the section by exploring the two-dimensional codes produced by traditional PCA and autoencoders. We'll want to show that autoencoders produce better visualizations. In particular, we'll want to show that autoencoders do a much better job of visually separating instances of different digit classes than PCA. We'll start by quickly covering the code we use to produce two-dimensional PCA codes:

```
from sklearn import decomposition
import input_data


mnist = input_data.read_data_sets("data/", one_hot=False)
```

```
pca = decomposition.PCA(n_components=2)
pca.fit(mnist.train.images)
pca_codes = pca.transform(mnist.test.images)
```

We first pull up the MNIST dataset. We've set the flag `one_hot=False` because we'd like the labels to be provided as integers instead of one-hot vectors (as a quick reminder, a one-hot vector representing an MNIST label would be a vector of size 10 with the i^{th} component set to one to represent digit i and the rest of the components set to zero). We use the commonly used machine learning library scikit-learn to perform the PCA, setting the `n_components=2` flat so that scikit-learn knows to generate two-dimensional codes. We can also reconstruct the original images from the two-dimensional codes and visualize the reconstructions:

```
from matplotlib import pyplot as plt

pca_recon = pca.inverse_transform(pca_codes[:1])
plt.imshow(pca_recon[0].reshape((28,28)), cmap=plt.cm.gray)
plt.show()
```

The code snippet shows how to visualize the first image in the test dataset, but we can easily modify the code to visualize any arbitrary subset of the dataset. Comparing the PCA reconstructions to the autoencoder reconstructions in Figure 6-10, it's quite clear that the autoencoder vastly outperforms PCA with two-dimensional codes. In fact, the PCA's performance is somewhat reminiscent of the autoencoder only five epochs into training. It has trouble distinguishing 5's from 3's and 8's, 0's from 8's, and 4's from 9's. Repeating the same experiment with 30-dimensional codes provides significant improvement to the PCA reconstructions, but they are still significantly worse than the 30-dimensional autoencoder.

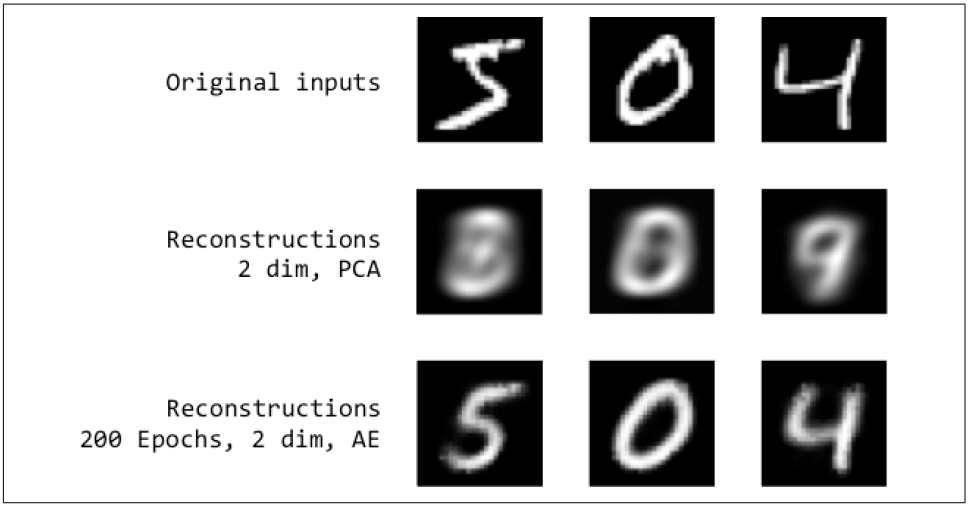


Figure 6-10. Comparing the reconstructions by both PCA and autoencoder side by side

Now, to complete the experiment, we must load up a saved TensorFlow model, retrieve the two-dimensional codes, and plot both the PCA and autoencoder codes. We're careful to rebuild the TensorFlow graph exactly how we set it up during training. We pass the path to the model checkpoint we saved during training as a command-line argument to the script. Finally, we use a custom plotting function to generate a legend and appropriately color data points of different digit classes:

```
import tensorflow as tf
import autoencoder_mnist as ae
import argparse

def scatter(codes, labels):
    colors = [
        ('#27ae60', 'o'),
        ('#2980b9', 'o'),
        ('#8e44ad', 'o'),
        ('#f39c12', 'o'),
        ('#c0392b', 'o'),
        ('#27ae60', 'x'),
        ('#2980b9', 'x'),
        ('#8e44ad', 'x'),
        ('#c0392b', 'x'),
        ('#f39c12', 'x'),
    ]
    for num in xrange(10):
        plt.scatter([codes[:,0][i] for i in xrange(len
                                (labels)) if labels[i] == num],
        [codes[:,1][i] for i in xrange(len(labels)) if
                                     labels[i] == num], 7,
        label=str(num), color = colors[num][0],
                                marker=colors[num][1])
    plt.legend()
    plt.show()


with tf.Graph().as_default():

        with tf.variable_scope("autoencoder_model"):

            x = tf.placeholder("float", [None, 784])
            phase_train = tf.placeholder(tf.bool)

            code = ae.encoder(x, 2, phase_train)

            output = ae.decoder(code, 2, phase_train)

            cost, train_summary_op = ae.loss(output, x)

            global_step = tf.Variable(0, name='global_step',
      trainable=False)
```

```
            train_op = ae.training(cost, global_step)

            eval_op, in_im_op, out_im_op, val_summary_op =
             ae.evaluate(output, x)

            saver = tf.train.Saver()

            sess = tf.Session()

            sess = tf.Session()
            saver = tf.train.Saver()
            saver.restore(sess, args.savepath[0])

            ae_codes= sess.run(code, feed_dict={x:
mnist.test.images, phase_train: True})

            scatter(ae_codes,
            mnist.test.labels)
            scatter(pca_codes, mnist.test.labels)
```

In the resulting visualization in Figure 6-11, it is extremely difficult to make out separable clusters in the two-dimensional PCA codes; the autoencoder has clearly done a spectacular job at clustering codes of different digit classes. This means that a simple machine learning model is going to be able to much more effectively classify data points consisting of autoencoder embeddings as compared to PCA embeddings.

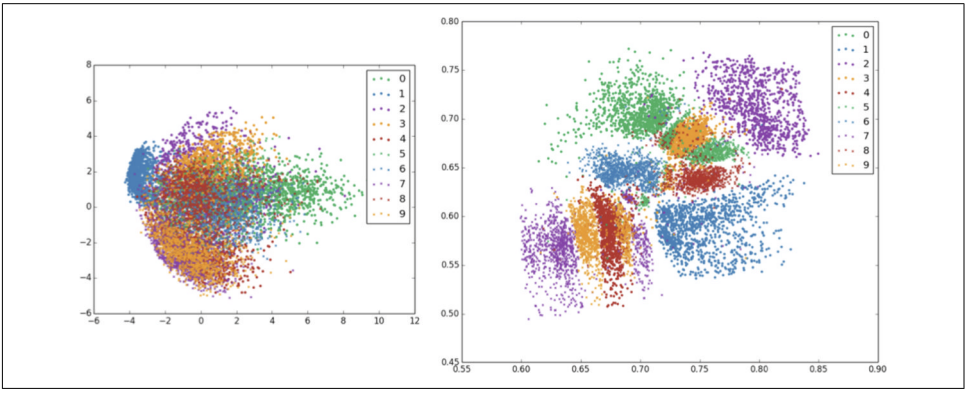


Figure 6-11. We visualize two-dimensional embeddings produced by PCA (left) and by an autoencoder (right). Notice that the autoencoder does a much better job of clustering codes of different digit classes.

In this section, we successfully set up and trained a feed-forward autoencoder and demonstrated that the resulting embeddings were superior to PCA, a classical dimensionality reduction method. In the next section, we'll explore a concept known as denoising, which acts as a form of regularization by making our embeddings more robust.

Denoising to Force Robust Representations

In this section, we'll explore an additional mechanism, known as *denoising*, to improve the ability of the autoencoder to generate embeddings that are resistant to noise. The human ability for perception is surprisingly resistant to noise. Take Figure 6-12, for example. Despite the fact that I've corrupted half of the pixels in each image, you still have no problem making out the digit. In fact, even easily confused digits (like the 2 and the 7) are still distinguishable.

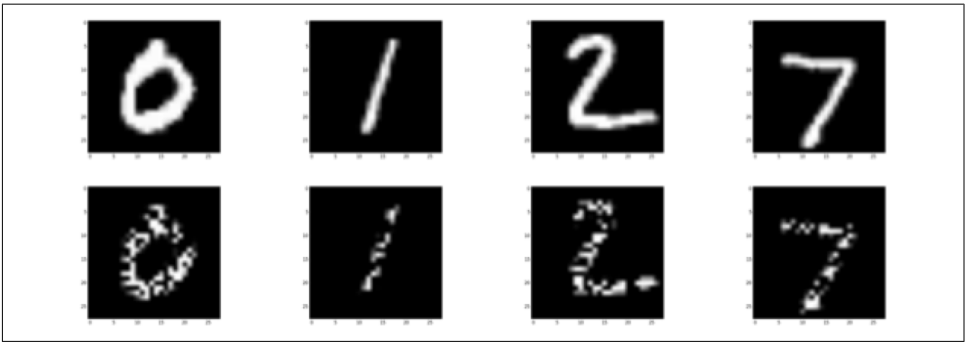


Figure 6-12. In the top row, we have original images from the MNIST dataset. In the bottom row, we've randomly blacked out half of the pixels. Despite the corruption, the digits in the bottom row are still identifiable by human perception.

One way to look at this phenomenon is probabilistically. Even if we're exposed to a random sampling of pixels from an image, if we have enough information, our brain is still capable of concluding the ground truth of what the pixels represent with maximal probability. Our mind is able to, quite literally, fill in the blanks to draw a conclusion. Even though only a corrupted version of a digit hits our retina, our brain is still able to reproduce the set of activations (i.e., the code or embedding) that we normally would use to represent the image of that digit. This is a property we might hope to enforce in our embedding algorithm, and it was first explored by Vincent et al. in 2008, when they introduced the *denoising autoencoder*.[2]

The basic principles behind denoising are quite simple. We corrupt some fixed percentage of the pixels in the input image by setting them to zero. Given an original input X, let's call the corrupted version $C(X)$. The denoising autoencoder is identical to the vanilla autoencoder except for one detail: the input to the encoder network is the corrupted $C(X)$ instead of X. In other words, the autoencoder is forced to learn a

2 Vincent, Pascal, et al. "Extracting and Composing Robust Features with Denoising Autoencoders." *Proceedings of the 25th International Conference on Machine Learning*. ACM, 2008.

code for each input that is resistant to the corruption mechanism and is able to interpolate through the missing information to recreate the original, uncorrupted image.

We can also think about this process more geometrically. Let's say we had a two-dimensional dataset with various labels. Let's take all of the data points in a particular category (i.e., with some fixed label), and call this subset of data points *S*. While any arbitrary sampling of points could end up taking any form while visualized, we presume that for real-life categories, there is some underlying structure that unifies all of the points in *S*. This underlying, unifying geometric structure is known as a *manifold*. The manifold is the shape that we want to capture when we reduce the dimensionality of our data; and as Alain and Bengio described in 2014, our autoencoder is implicitly learning this manifold as it learns how to reconstruct data after pushing it through a bottleneck (the code layer).[3] The autoencoder must figure out whether a point belongs to one manifold or another when trying to generate a reconstruction of an instance with potentially different labels.

As an illustration, let's consider the scenario in Figure 6-13, where the points in *S* are a simple low-dimensional manifold (in this case, a circle which is colored black in the diagram). In part A, we see our data points in *S* (black x's) and the manifold that best describes them. We also observe an approximation of our corruption operation. Specifically, the red arrow and solid red circle demonstrate all the ways in which the corruption could possibly move or modify a data point. Given that we are applying this corruption operation to every data point (i.e., along the entire manifold), this corruption operation artificially expands the dataset to not only include the manifold but also all of the points in space around the manifold, up to a maximum margin of error. This margin is demonstrated by the dotted red circles in A, and the dataset expansion is illustrated by the red x's in part B. Finally the autoencoder is forced to learn to collapse all of the data points in this space back to the manifold. In other words, by learning which aspects of a data point are generalizable, broad strokes and which aspects are "noise," the denoising autoencoder learns to approximate the underlying manifold of *S*.

3 Bengio, Yoshua, et al. "Generalized Denoising Auto-Encoders as Generative Models." *Advances in Neural Information Processing Systems*. 2013.

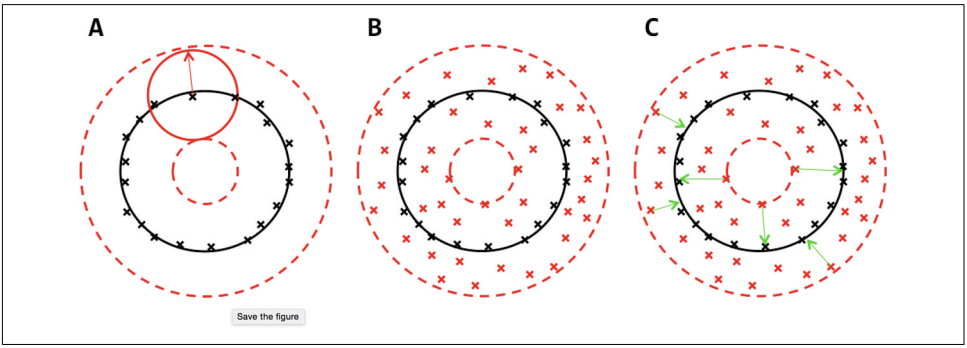


Figure 6-13. The denoising objective enables our model to learn the manifold (black circle) by learning to map corrupted data (red x's) to uncorrupted data (black x's) by minimizing the error (green arrows) between their representations

With the philosophical motivations of denoising in mind, we can now make a small modification to our autoencoder script to build a denoising autoencoder:

```
def corrupt_input(x):
    corrupting_matrix = tf.random_uniform(shape=tf.shape(x),
                        minval=0,maxval=2,dtype=tf.int32)
    return x * tf.cast(corrupting_matrix, tf.float32)

x = tf.placeholder("float", [None, 784]) # mnist data image of
                                         # shape 28*28=784
corrupt = tf.placeholder(tf.float32)
phase_train = tf.placeholder(tf.bool)

c_x = (corrupt_input(x) * corrupt) + (x * (1 - corrupt))
```

This code snippet corrupts the input if the `corrupt` placeholder is equal to 1, and it refrains from corrupting the input if the `corrupt` placeholder tensor is equal to 0. After making this modification, we can rerun our autoencoder, resulting in the reconstructions shown in Figure 6-14. It's quite apparent that the denoising autoencoder has faithfully replicated our incredible human ability to fill in the missing pixels.

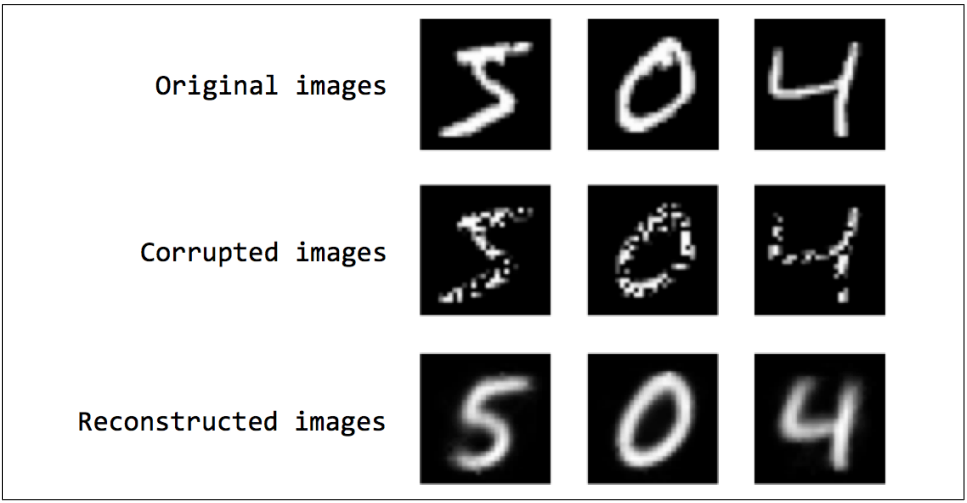


Figure 6-14. We apply a corruption operation to the dataset and train a denoising autoencoder to reconstruct the original, uncorrupted images

Sparsity in Autoencoders

One of the most difficult aspects of deep learning is a problem known as *interpretability*. Interpretability is a property of a machine learning model that measures how easy it is to inspect and explain its process and/or output. Deep models are generally very difficult to interpret because of the nonlinearities and massive numbers of parameters that make up a model. While deep models are generally more accurate, a lack of interpretability often hinders their adoption in highly valuable, but highly risky, applications. For example, if a machine learning model is predicting that a patient has or does not have cancer, the doctor will likely want an explanation to confirm the model's conclusion.

We can address one aspect of interpretability by exploring the characteristics of the output of an autoencoder. In general, an autoencoder's representations are dense, and this has implications with respect to how the representation changes as we make coherent modifications to the input. Consider the situation in Figure 6-15.

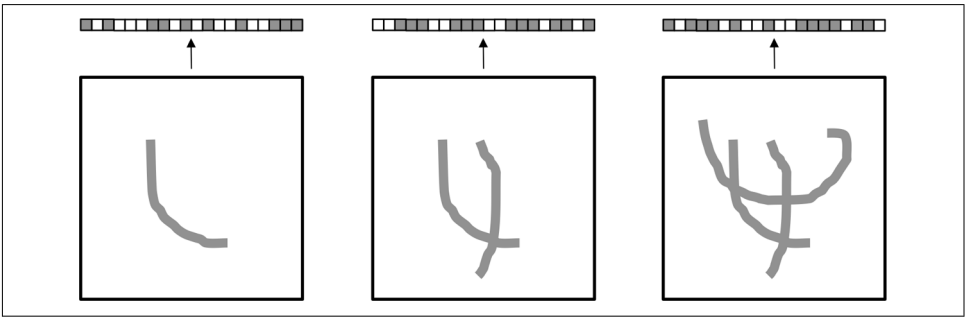


Figure 6-15. The activations of a dense representation combine and overlay information from multiple features in ways that are difficult to interpret

The autoencoder produces a *dense* representation, that is, the representation of the original image is highly compressed. Because we only have so many dimensions to work with in the representation, the activations of the representation combine information from multiple features in ways that are extremely difficult to disentangle. The result is that as we add components or remove components, the output representation changes in unexpected ways. It's virtually impossible to interpret how and why the representation is generated in the way it is.

The ideal outcome for us is if we can build a representation where there is a 1-to-1 correspondence, or close to a 1-to-1 correspondence, between high-level features and individual components in the code. When we are able to achieve this, we get very close to the system described in Figure 6-16. Part A of the figure shows how the representation changes as we add and remove components, and part B color-codes the correspondence between strokes and the components in the code. In this setup, it's quite clear how and why the representation changes—the representation is very clearly the sum of the individual strokes in the image.

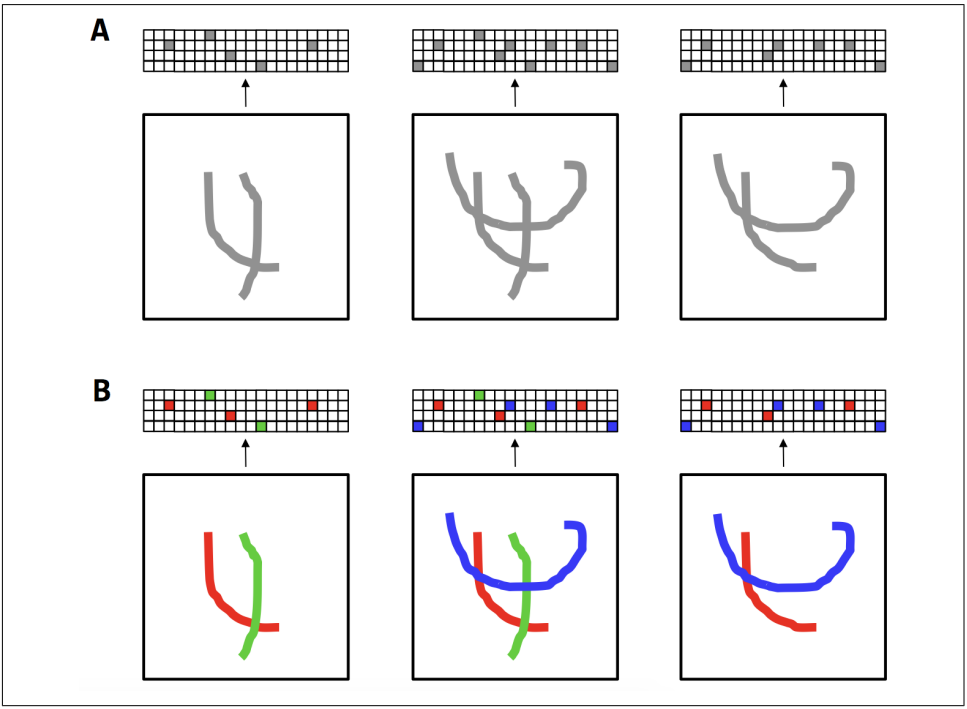


Figure 6-16. With the right combination of space and sparsity, a representation is more interpretable. In A, we show how activations in the representation change with the addition and removal of strokes. In B, we color-code the activations that correspond to each stroke to highlight our ability to interpret how a stroke affects the representation.

While this is the ideal outcome, we'll have to think through what mechanisms we can leverage to enable this interpretability in the representation. The issue here is clearly the bottlenecked capacity of the code layer; but unfortunately, increasing the capacity of the code layer alone is not sufficient. In the medium case, while we can increase the size of the code layer, there is no mechanism that prevents each individual feature picked up by the autoencoder from affecting a large fraction of the components with smaller magnitudes. In the more extreme case, where the features that are picked up are more complex and therefore more bountiful, the capacity of the code layer may be even larger than the dimensionality of the input. In this case, the code layer has so much capacity that the model could quite literally perform a "copy" operation where the code layer learns no useful representation.

What we really want is to force the autoencoder to utilize as few components of the representation vector as possible, while still effectively reconstructing the input. This is very similar to the rationale behind using regularization to prevent overfitting in simple neural networks, as we discussed in Chapter 2, except we want as many components to be zero (or extremely close to zero) as possible. As in Chapter 2, we'll ach-

ieve this by modifying the objective function with a sparsity penalty, which increases the cost of any representation that has a large number of nonzero components:

$$E_{\text{Sparse}} = E + \beta \cdot \text{SparsityPenalty}$$

The value of β determines how strongly we favor sparsity at the expense of generating better reconstructions. For the mathematically inclined, you would do this by treating the values of each of the components of every representation as the outcome of a random variable with an unknown mean. We would then employ a measure of divergence comparing the distribution of observations of this random variable (the values of each component) and the distribution of a random variable whose mean is known to be 0. A measure that is often used to this end is the Kullback-Leibler (often referred to as KL) divergence. Further discussion on sparsity in autoencoders is beyond the scope of this text, but they are covered by Ranzato et al. (2007[4] and 2008[5]). More recently, the theoretical properties and empirical effectiveness of introducing an intermediate function before the code layer that zeroes out all but k of the maximum activations in the representation were investigated by Makhzani and Frey (2014).[6] These *k-Sparse autoencoders* were shown to be just as effective as other mechanisms of sparsity despite being shockingly simple to implement and understand (as well as computationally more efficient).

This concludes our discussion of autoencoders. We've explored how we can use autoencoders to find strong representations of data points by summarizing their content. This mechanism of dimensionality reduction works well when the independent data points are rich and contain all of the relevant information pertaining to their structure in their original representation. In the next section, we'll explore strategies that we can use when the main source of information is in the context of the data point instead of the data point itself.

When Context Is More Informative than the Input Vector

In the previous sections of this chapter, we've mostly focused on the concept of dimensionality reduction. In dimensionality reduction, we generally have rich inputs which contain lots of noise on top of the core, structural information that we care about. In these situations, we want to extract this underlying information while ignoring the variations and noise that are extraneous to this fundamental understanding of the data.

---

4 Ranzato, Marc'Aurelio, et al. "Efficient Learning of Sparse Representations with an Energy-Based Model." *Proceedings of the 19th International Conference on Neural Information Processing Systems*. MIT Press, 2006.

5 Ranzato, Marc'Aurelio, and Martin Szummer. "Semi-supervised Learning of Compact Document Representations with Deep Networks." *Proceedings of the 25th International Conference on Machine Learning*. ACM, 2008.

6 Makhzani, Alireza, and Brendan Frey. "k-Sparse Autoencoders." *arXiv preprint arXiv*:1312.5663 (2013).

In other situations, we have input representations that say very little at all about the content that we are trying to capture. In these situations, our goal is not to extract information, but rather, to gather information from context to build useful representations. All of this probably sounds too abstract to be useful at this point, so let's concretize these ideas with a real example.

Building models for language is a tricky business. The first problem we have to overcome when building language models is finding a good way to represent individual words. At first glance, it's not entirely clear how one builds a good representation. Let's start with the naive approach, considering the illustrative example in Figure 6-17.

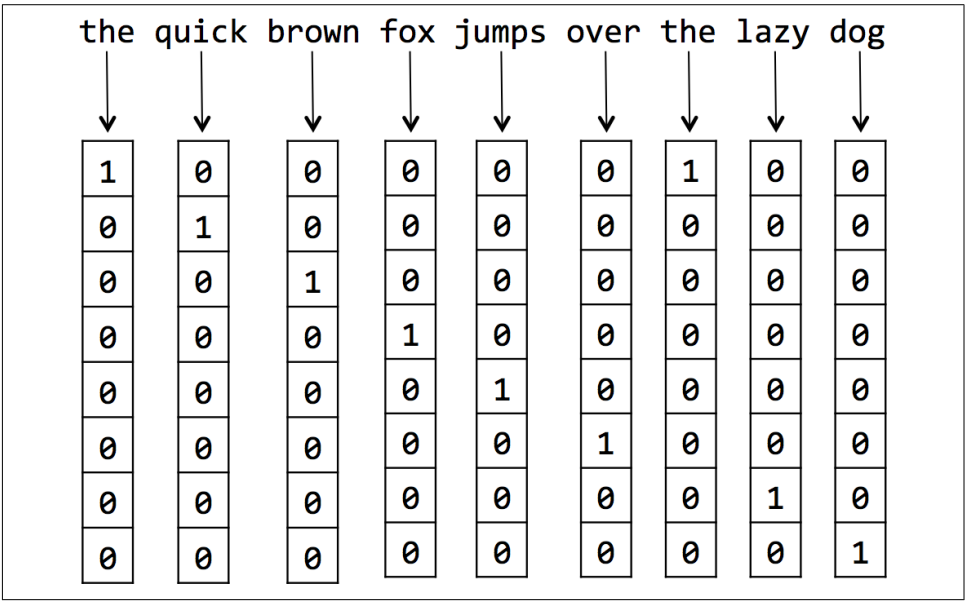


Figure 6-17. An example of generating one-hot vector representations for words using a simple document

If a document has a vocabulary V with $|V|$ words, we can represent the words with one-hot vectors. In other words, we have $|V|$-dimensional representation vectors, and we associate each unique word with an index in this vector. To represent unique word w_i, we set the i^{th} component of the vector to be 1, and zero out all of the other components.

However, this representation scheme seems rather arbitrary. This vectorization does not make similar words into similar vectors. This is problematic, because we'd like our models to know that the words "jump" and "leap" have very similar meanings. Similarly we'd like our models to know when words are verbs or nouns or prepositions. The naive one-hot encoding of words to vectors does not capture any of these

characteristics. To address this challenge, we'll need to find some way of discovering these relationships and encoding this information into a vector.

It turns out that one way to discover relationships between words is by analyzing their surrounding context. For example, synonyms such as "jump" and "leap" both can be used interchangeably in their respective contexts. In addition, both words generally appear when a subject is performing the action over a direct object. We use this principle all the time when we run across new vocabulary while reading. For example, if we read the sentence "The warmonger argued with the crowd," we can immediately draw conclusions about the word "warmonger" even if we don't already know the dictionary definition. In this context, "warmonger" precedes a word we know to be a verb, which makes it likely that "warmonger" is a noun and the subject of this sentence. Also, the "warmonger" is "arguing," which might imply that a "warmonger" is generally a combative or argumentative individual. Overall, as illustrated in Figure 6-18, by analyzing the context (i.e., a fixed window of words surrounding a target word), we can quickly surmise the meaning of the word.

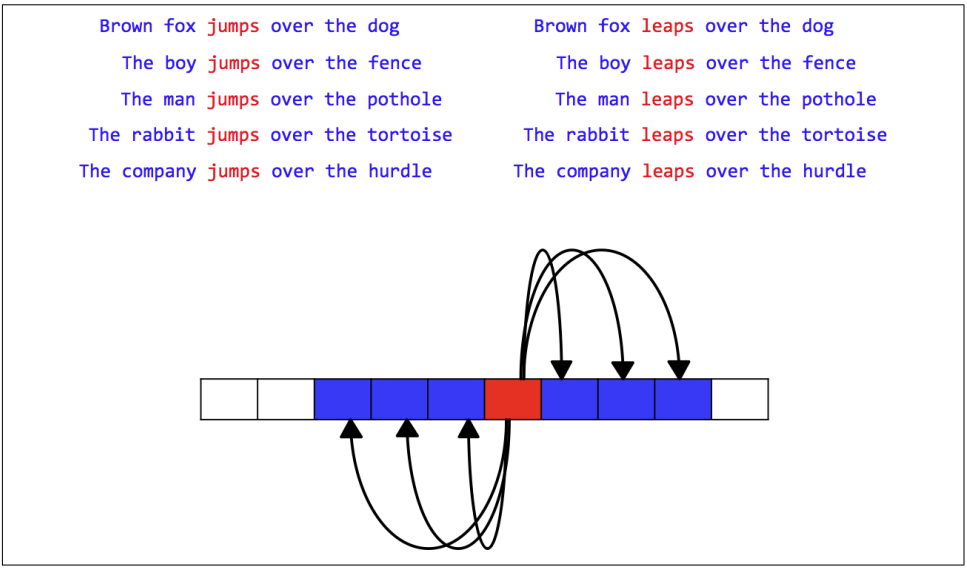


Figure 6-18. We can identify words with similar meanings based on their contexts. For example, the words "jumps" and "leaps" should have similar vector representations because they are virtually interchangeable. Moreover, we can draw conclusions about what the words "jumps" and "leaps" mean just by looking at the words around them.

It turns out we can use the same principles we used when building the autoencoder to build a network that builds strong, distributed representations. Two strategies are shown in Figure 6-19. One possible method (shown in A) passes the target through an encoder network to create an embedding. Then we have a decoder network take

this embedding; but instead of trying to reconstruct the original input as we did with the autoencoder, the decoder attempts to construct a word from the context. The second possible method (shown in B) does exactly the reverse: the encoder takes a word from the context as input, producing the target.

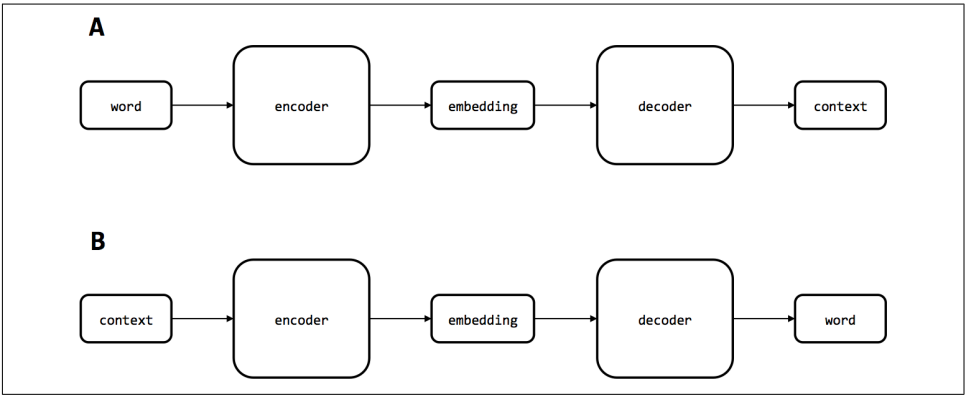


Figure 6-19. General architectures for designing encoders and decoders that generate embeddings by mapping words to their respective contexts (A) or vice versa (B)

In the next section, we'll describe how we use this strategy (along with some slight modifications for performance) to produce word embeddings in practice.

The Word2Vec Framework

Word2Vec, a framework for generating word embeddings, was pioneered by Mikolov et al. The original paper detailed two strategies for generating embeddings, very similar to the two strategies for encoding context we discussed in the previous section.

The first flavor of Word2Vec Mikolov et al. introduced was the *Continuous Bag of Words* (CBOW) model.[7] This model is much like strategy B from the previous section. The CBOW model used the encoder to create an embedding from the full context (treated as one input) and predict the target word. It turns out this strategy works best for smaller datasets, an attribute that is further discussed in the original paper.

The second flavor of Word2Vec is the *Skip-Gram model*, introduced by Mikolov et al.[8]. The Skip-Gram model does the inverse of CBOW, taking the target word as an

7 Mikolov, Tomas, et al. "Distributed Representations of Words and Phrases and their Compositionality." *Advances in Neural Information Processing Systems*. 2013.

8 Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space" *ICLR Workshop*, 2013.

input, and then attempting to predict one of the words in the context. Let's walk through a toy example to explore what the dataset for a Skip-Gram model looks like.

Consider the sentence "the boy went to the bank." If we broke this sentence down into a sequence of (context, target) pairs, we would obtain [([the, went], boy), ([boy, to], went), ([went, the], to), ([to, bank], the)]. Taking this a step further, we have to split each (context, target) pair into (input, output) pairs where the input is the target and the output is one of the words from the context. From the first pair ([the, went], boy), we would generate the two pairs (boy, the) and (boy, went). We continue to apply this operation to every (context, target) pair to build our dataset. Finally, we replace each word with its unique index $i \in \{0, 1, \ldots, |V| - 1\}$ corresponding to its index in the vocabulary.

The structure of the encoder is surprisingly simple. It is essentially a lookup table with $|V|$ rows, where the i^{th} row is the embedding corresponding to the i^{th} vocabulary word. All the encoder has to do is take the index of the input word and output the appropriate row in the lookup table. This an efficient operation because on a GPU, this operation can be represented as a product of the transpose of the lookup table and the one-hot vector representing the input word. We can implement this simply in TensorFlow with the following TensorFlow function:

```
tf.nn.embedding_lookup(params, ids, partition_strategy='mod',
                       name=None, validate_indices=True)
```

Where `params` is the embedding matrix, and `ids` is a tensor of indices we want to look up. For information on optional parameters, we refer the curious reader to the Tensorflow API documentation.[9]

The decoder is slightly trickier because we make some modifications for performance. The naive way to construct the decoder would be to attempt to reconstruct the one-hot encoding vector for the output, which we could implement with a run-of-the-mill feed-forward layer coupled with a softmax. The only concern is that it's inefficient because we have to produce a probability distribution over the whole vocabulary space.

To reduce the number of parameters, Mikolov et al. used a strategy for implementing the decoder known as noise-contrastive estimation (NCE). The strategy is illustrated in Figure 6-20.

9 *https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup*

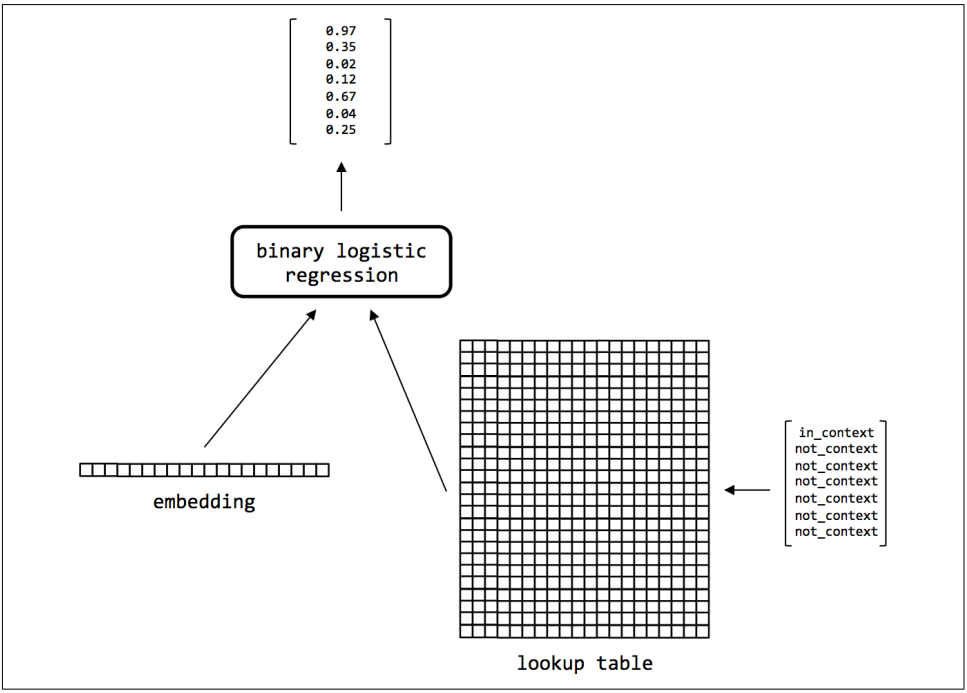


Figure 6-20. An illustration of how noise-contrastive estimation works. A binary logistic regression compares the embedding of the target with the embedding of a context word and randomly sampled noncontext words. We construct a loss function describing how effectively the embeddings enable identification of words in the context of the target versus words outside the context of the target.

The NCE strategy uses the lookup table to find the embedding for the output, as well as embeddings for random selections from the vocabulary that are not in the context of the input. We then employ a binary logistic regression model that, one at a time, takes the input embedding and the embedding of the output or random selection, and then outputs a value between 0 to 1 corresponding to the probability that the comparison embedding represents a vocabulary word present in the input's context. We then take the sum of the probabilities corresponding to the noncontext comparisons and subtract the probability corresponding to the context comparison. This value is the objective function that we want to minimize (in the optimal scenario where the model has perfect performance, the value will be -1). Implementing NCE in TensorFlow utilizes the following code snippet:

```
tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled,
               num_classes, num_true=1, sampled_values=None,
               remove_accidental_hits=False, partition_strategy=
                       'mod',
               name='nce_loss')
```

The `weights` should have the same dimensions as the embedding matrix, and the `biases` should be a tensor with size equal to the vocabulary. The `inputs` are the results from the embedding lookup, `num_sampled` is the number of negative samples we use to compute the NCE, and `num_classes` is the vocabulary size.

While Word2Vec is admittedly not a deep machine learning model, we discuss it here for many reasons. First, it thematically represents a strategy (finding embeddings using context) that generalizes to many deep learning models. When we learn about models for sequence analysis in Chapter 7, we'll see this strategy employed for generating skip-thought vectors to embed sentences. Moreover, when we start building more and more models for language starting in Chapter 7, we'll find that using Word2Vec embeddings instead of one-hot vectors to represent words will yield far superior results.

Now that we understand how to architect the Skip-Gram model and its importance, we can start implementing it in TensorFlow.

Implementing the Skip-Gram Architecture

To build the dataset for our Skip-Gram model, we'll utilize a modified version of the TensorFlow Word2Vec data reader in `input_word_data.py`. We'll start off by setting a couple of important parameters for training and regularly inspecting our model. Of particular note, we employ a minibatch size of 32 examples and train for 5 epochs (full passes through the dataset). We'll utilize embeddings of size 128. We'll use a context window of five words to the left and to the right of each target word, and sample four context words from this window. Finally, we'll use 64 randomly chosen non-context words for NCE.

Implementing the embedding layer is not particularly complicated. We merely have to initialize the lookup table with a matrix of values:

```
def embedding_layer(x, embedding_shape):
    with tf.variable_scope("embedding"):
        embedding_init = tf.random_uniform(embedding_shape,
     -1.0, 1.0)
        embedding_matrix = tf.get_variable("E",
                                   initializer=embedding_init)
        return tf.nn.embedding_lookup(embedding_matrix, x),
                                         embedding_matrix
```

We utilize TensorFlow's built-in `tf.nn.nce_loss` to compute the NCE cost for each training example, and then compile all of the results in the minibatch into a single measurement:

```
def noise_contrastive_loss(embedding_lookup, weight_shape,
                           bias_shape, y):
    with tf.variable_scope("nce"):
```

```
        nce_weight_init = tf.truncated_normal(weight_shape,
                                              stddev=1.0/(
                                        weight_shape[1])**0.5)
        nce_bias_init = tf.zeros(bias_shape)
        nce_W = tf.get_variable("W",
                    initializer=nce_weight_init)
        nce_b = tf.get_variable("b", initializer=nce_bias_init)

        total_loss = tf.nn.nce_loss(nce_W, nce_b,
                    embedding_lookup,
                                                y, neg_size,
                                    data.vocabulary_size)
        return tf.reduce_mean(total_loss)
```

Now that we have our objective function expressed as a mean of the NCE costs, we set up the training as usual. Here, we follow in the footsteps of Mikolov et al. and employ stochastic gradient descent with a learning rate of 0.1:

```
def training(cost, global_step):
    with tf.variable_scope("training"):
        summary_op = tf.scalar_summary("cost", cost)
        optimizer = tf.train.GradientDescentOptimizer(
                                learning_rate)
        train_op = optimizer.minimize(
                     cost, global_step=global_step)
        return train_op, summary_op
```

We also inspect the model regularly using a validation function, which normalizes the embeddings in the lookup table and uses cosine similarity to compute distances for a set of validation words from all other words in the vocabulary:

```
def validation(embedding_matrix, x_val):
    norm = tf.reduce_sum(embedding_matrix**2, 1,
                                   keep_dims=True)**0.5
    normalized = embedding_matrix/norm
    val_embeddings = tf.nn.embedding_lookup(normalized, x_val)
    cosine_similarity = tf.matmul(val_embeddings, normalized,
                                      transpose_b=True)
    return normalized, cosine_similarity
```

Putting all of these components together, we're finally ready to run the Skip-Gram model. We skim over this portion of the code because it is very similar to how we constructed models in the past. The only difference is the additional code during the inspection step. We randomly select 20 validation words out of the 500 most common words in our vocabulary of 10,000 words. For each of these words, we use the cosine similarity function we built to find the nearest neighbors:

```
if __name__ == '__main__':

    with tf.Graph().as_default():

        with tf.variable_scope("skipgram_model"):

            x = tf.placeholder(tf.int32, shape=[batch_size])
            y = tf.placeholder(tf.int32, [batch_size, 1])
            val = tf.constant(val_examples, dtype=tf.int32)
            global_step = tf.Variable(0, name='global_step',
                                      trainable=False)

            e_lookup, e_matrix =
               embedding_layer(x,
               [data.vocabulary_size, embedding_size])

            cost = noise_contrastive_loss(e_lookup,
                                [data.vocabulary_size,
                                  embedding_size],
                                  [data.vocabulary_size], y)

            train_op, summary_op = training(cost, global_step)

            val_op = validation(e_matrix, val)

            sess = tf.Session()

            train_writer = tf.train.SummaryWriter(
                "skipgram_logs/", graph=sess.graph)

            init_op = tf.initialize_all_variables()

            sess.run(init_op)

            step = 0
            avg_cost = 0

            for epoch in xrange(training_epochs):
                for minibatch in xrange(batches_per_epoch):

                    step +=1

                    mbatch_x, mbatch_y = data.generate_batch(
                            batch_size,
                       num_skips, skip_window)
                    feed_dict = {x : mbatch_x, y : mbatch_y}

                    _, new_cost, train_summary = sess.run([
                                    train_op, cost,
                                    summary_op],
                                    feed_dict=feed_dict)
                    train_writer.add_summary(train_summary,
```

```
                        sess.run(global_step))
                # Compute average loss
                avg_cost += new_cost/display_step

                if step % display_step == 0:
                    print "Elapsed:", str(step), "batches.
                                Cost =",
                          "{:.9f}".format(avg_cost)
                    avg_cost = 0

                if step % val_step == 0:
                    _, similarity = sess.run(val_op)
                    for i in xrange(val_size):
                        val_word = data.reverse_dictionary
                                        [val_examples[i]]
                        neighbors = (-similarity[
                                       i, :]).argsort()
                                         [1:top_match+1]
                        print_str = "Nearest neighbor of
                                 %s:"
                                          % val_word
                        for k in xrange(top_match):
                            print_str += " %s," %
                        data.reverse_dictionary[
                          neighbors[k]]
                        print print_str[:-1]

        final_embeddings, _ = sess.run(val_op)
```

The code starts to run, and we can begin to see how the model evolves over time. At the beginning, the model does a poor job of embedding (as is apparent from the inspection step). However, by the time training completes, the model has clearly found representations that effectively capture the meanings of individual words:

```
ancient: egyptian, cultures, mythology, civilization, etruscan,
greek, classical, preserved


however: but, argued, necessarily, suggest, certainly, nor,
believe, believed


type: typical, kind, subset, form, combination, single,
description, meant


white: yellow, black, red, blue, colors, grey, bright, dark


system: operating, systems, unix, component, variant, versions,
version, essentially
```

energy: kinetic, amount, heat, gravitational, nucleus, radiation, particles, transfer

world: ii, tournament, match, greatest, war, ever, championship, cold

y: z, x, n, p, f, variable, mathrm, sum,

line: lines, ball, straight, circle, facing, edge, goal, yards,

among: amongst, prominent, most, while, famous, particularly, argue, many

image: png, jpg, width, images, gallery, aloe, gif, angel

kingdom: states, turkey, britain, nations, islands, namely, ireland, rest

long: short, narrow, thousand, just, extended, span, length, shorter

through: into, passing, behind, capture, across, when, apart, goal

i: you, t, know, really, me, want, myself, we

source: essential, implementation, important, software, content, genetic, alcohol, application

because: thus, while, possibility, consequently, furthermore, but, certainly, moral

eight: six, seven, five, nine, one, four, three, b

french: spanish, jacques, pierre, dutch, italian, du, english, belgian

```
written: translated, inspired, poetry, alphabet, hebrew,
letters, words, read
```

While not perfect, there are some strikingly meaningful clusters captured here. Numbers, countries, and cultures are clustered close together. The pronoun "I" is clustered with other pronouns. The word "world" is interestingly close to both "championship" and "war." And the word "written" is found to be very similar to "translated," "poetry," "alphabet," "letters," and "words."

Finally, we conclude this section by visualizing our word embeddings in Figure 6-21. To display our 128-dimensional embeddings in 2-dimensional space, we'll use a visualization method known as t-SNE. If you'll recall, we also used t-SNE in Chapter 5 to visualize the relationships between images in ImageNet. Using t-SNE is quite simple, as it has a built-in function in the commonly used machine learning library scikit-learn.

We can construct the visualization using the following code:

```
tsne = TSNE(perplexity=30, n_components=2, init='pca',
            n_iter=5000)
plot_embeddings = np.asfarray(final_embeddings[:plot_num,:],
                              dtype='float')
low_dim_embs = tsne.fit_transform(plot_embeddings)
labels = [reverse_dictionary[i] for i in xrange(plot_only)]
data.plot_with_labels(low_dim_embs, labels)
```

For a more detailed exploration of the properties of word embeddings and interesting patterns (verb tenses, countries and capitals, analogy completion, etc.), we refer the curious reader to the original Mikolov et al. paper.

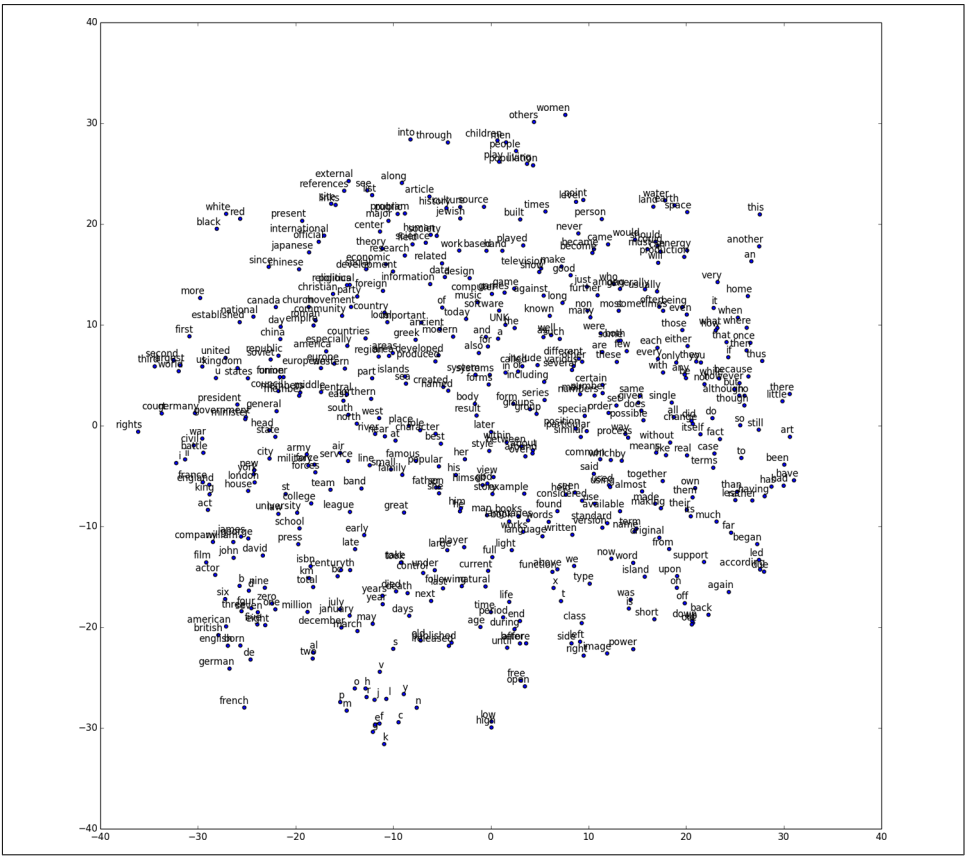


Figure 6-21. Visualization of our Skip-Gram embeddings using t-SNE. We notice that similar concepts are closer together than disparate concepts, indicating that our embeddings encode meaningful information about the functions and definitions of individual words.

Summary

In this chapter, we explored various methods in representation learning. We learned about how we can perform effective dimensionality reduction using autoencoders. We also learned about denoising and sparsity, which augment autoencoders with useful properties. After discussing autoencoders, we shifted our attention to representation learning when context of an input is more informative than the input itself. We learned how to generate embeddings for English words using the Skip-Gram model, which will prove useful as we explore deep learning models for understanding language. In the next chapter, we will build on this tangent to analyze language and other sequences using deep learning.

CHAPTER 7

Models for Sequence Analysis

Mostafa Samir[1] *and Surya Bhupatiraju*

Analyzing Variable-Length Inputs

Up until now, we've only worked with data with fixed sizes: images from MNIST, CIFAR-10, and ImageNet. These models are incredibly powerful, but there are many situations in which fixed-length models are insufficient. The vast majority of interactions in our daily lives require a deep understanding of sequences—whether it's reading the morning newspaper, making a bowl of cereal, listening to the radio, watching a presentation, or deciding to execute a trade on the stock market. To adapt to variable-length inputs, we'll have to be a little bit more clever about how we approach designing deep learning models.

In Figure 7-1, we illustrate how our feed-forward neural networks break when analyzing sequences. If the sequence is the same size as the input layer, the model can perform as we expect it to. It's even possible to deal with smaller inputs by padding zeros to the end of the input until it's the appropriate length. However, the moment the input exceeds the size of the input layer, naively using the feedforward network no longer works.

1 *https://mostafa-samir.github.io/*

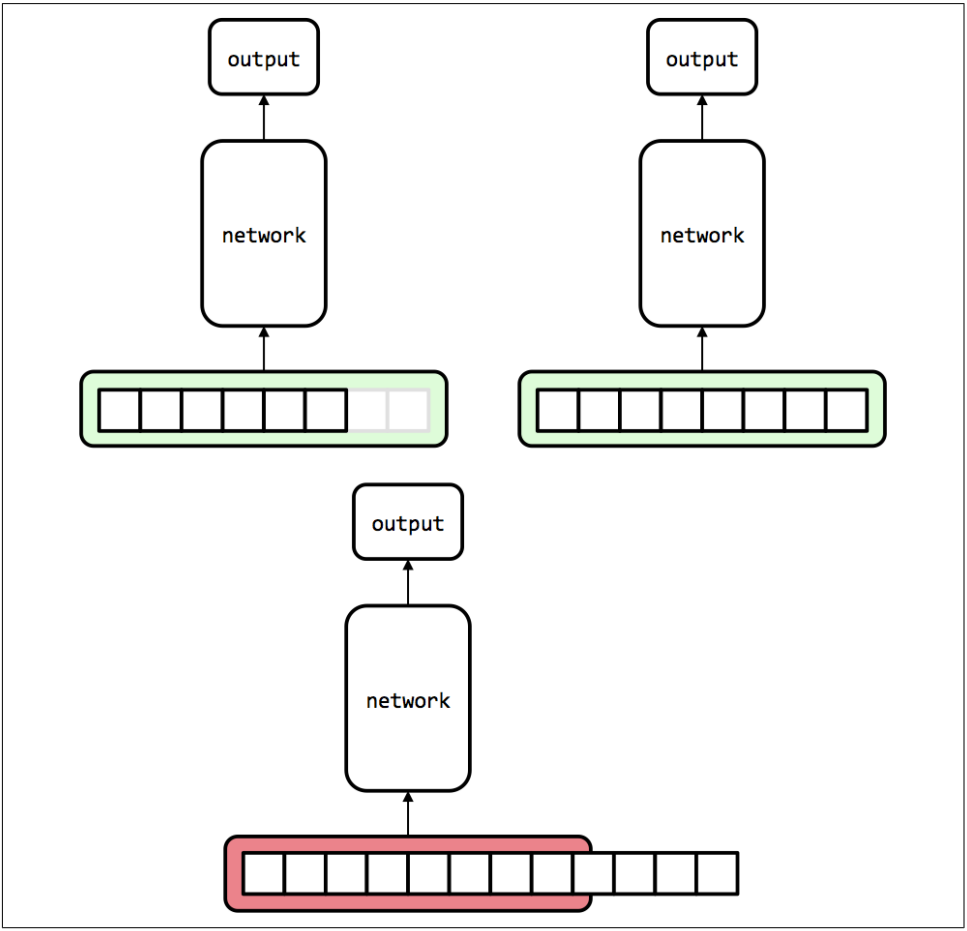


Figure 7-1. Feed-forward networks thrive on fixed input size problems. Zero padding can address the handling of smaller inputs, but when naively utilized, these models break when inputs exceed the fixed input size.

Not all hope is lost, however. In the next couple of sections, we'll explore several strategies we can leverage to "hack" feedfoward networks to handle sequences. Later in the chapter, we'll analyze the limitations of these hacks and discuss new architectures to address them. Finally, we will conclude the chapter by discussing some of the most advanced architectures explored to date to tackle some of the most difficult challenges in replicating human-level logical reasoning and cognition over sequences.

Tackling seq2seq with Neural N-Grams

In this section, we'll begin exploring a feed-forward neural network architecture that can process a body of text and produce a sequence of part-of-speech (POS) tags. In other words, we want to appropriately label each word in the input text as a noun, verb, preposition, and so on. An example of this is shown in Figure 7-2. While it's not the same complexity as building an AI that can answer questions after reading a story, it's a solid first step toward developing an algorithm that can understand the meaning of how words are used in a sentence. This problem is also interesting because it is an instance of a class of problems known as *seq2seq*, where the goal is to transform an input sequence into a corresponding output sequence. Other famous seq2seq problems include translating text between languages (which we will tackle later in this chapter), text summarization, and transcribing speech to text.

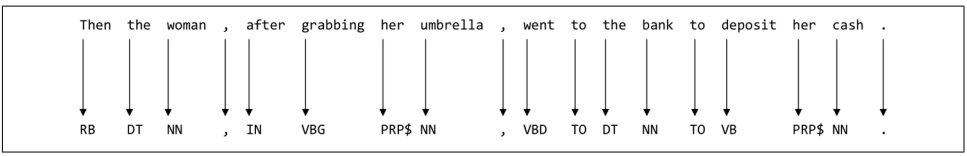


Figure 7-2. An example of an accurate POS parse of an English sentence

As we discussed, it's not obvious how we might take a body of text all at once to predict the full sequence of POS tags. Instead, we leverage a trick that is akin to the way we developed distributed vector representations of words in the previous chapter. The key observation is this: *it is not necessary to take into account long-term dependencies to predict the POS of any given word.*

The implication of this observation is that instead of using the whole sequence to predict all of the POS tags simultaneously, we can predict each POS tag one at a time by using a fixed-length subsequence. In particular, we utilize the subsequence starting from the word of interest and extending *n* words into the past. This *neural n-gram strategy* is depicted in Figure 7-3.

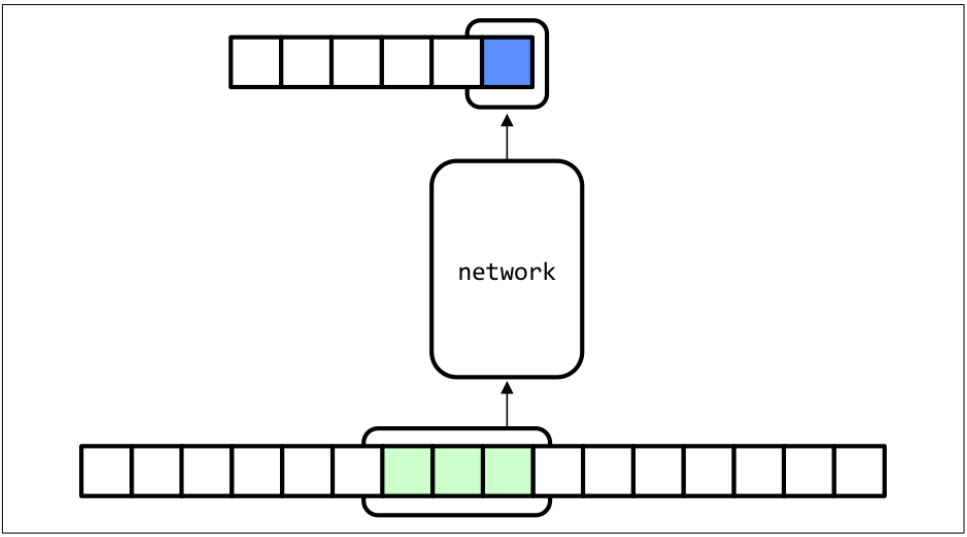


Figure 7-3. Using a feed-forward network to perform seq2seq when we can ignore long-term dependencies

Specifically, when we predict the POS tag for the i^{th} word in the input, we utilize the the $i - n + 1^{st}, i - n + 2^{nd}, \ldots, i^{th}$ words as the input. We'll refer to this subsequence as the *context window*. In order to process the entire text, we'll start by positioning the network at the very beginning of the text. We'll then proceed to move the network's context window one word at a time, predicting the POS tag of the rightmost word, until we reach the end of the input.

Leveraging the word embedding strategy from last chapter, we'll also use condensed representations of the words instead of one-hot vectors. This will allow us to reduce the number of parameters in our model and make learning faster.

Implementing a Part-of-Speech Tagger

Now that we have a strong understanding of the POS network architecture, we can dive into the implementation. On a high level, the network consists of an input layer that leverages a 3-gram context window. We'll utilize word embeddings that are 300-dimensional, resulting in a context window of size 900. The feed-forward network will have two hidden layers of size 512 neurons and 256 neurons, respectively. Finally, the output layer will be a softmax calculating the probability distribution of the POS tag output over a space of 44 possible tags. As usual, we'll use the Adam optimizer with our default hyperparameter settings, train for a total of 1,000 epochs, and leverage batch-normalization for regularization.

The actual network is extremely similar to networks we've implemented in the past. Rather, the tricky part of building the POS tagger is in preparing the dataset. We'll leverage pretrained word embeddings generated from Google News.[2] It includes vectors for 3 million words and phrases and was trained on roughly 100 billion words. We can use the `gensim` Python package to read the dataset. We use `pip` to install the package:

```
$ pip install gensim
```

We can subsequently load these vectors into memory using the following command:

```
from gensim.models import Word2Vec

model = Word2Vec.load_word2vec_format('/path/to/googlenews.bin',
                                      binary=True)
```

The issue with this operation, however, is that it's incredibly slow (it can take up to an hour, depending on the specs of your machine). To avoid loading the full dataset into memory every single time we run our program, especially while debugging code or experimenting with different hyperparameters, we cache the relevant subset of the vectors to disk using a lightweight database known as LevelDB.[3] To build the appropriate Python bindings (which allow us to interact with a LevelDB instance from Python), we simply use the following command:

```
$ pip install leveldb
```

As we mentioned, the `gensim` model contains three million words, which is larger than our dataset. For the sake of efficiency, we'll selectively cache word vectors for words in our dataset and discard everything else. To figure out which words we'd like to cache, let's download the POS dataset from the CoNLL-2000 task.[4]

```
$ wget http://www.cnts.ua.ac.be/conll2000/chunking/train.txt.gz
  -O - | gunzip |
  cut -f1,2 -d" " > pos.train.txt

$ wget http://www.cnts.ua.ac.be/conll2000/chunking/test.txt.gz
  -O - | gunzip |
  cut -f1,2 -d " " > pos.test.txt
```

The dataset consists of contiguous text that is formatted as a sequence of rows, where the first element is a word and the second element is the corresponding part of speech. Here are the first several lines of the training dataset:

```
Confidence NN
in IN
```

2 Google News download link: *https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit*

3 *http://leveldb.org/*

4 *http://www.cnts.ua.ac.be/conll2000/chunking/*

```
the DT
pound NN
is VBZ
widely RB
expected VBN
to TO
take VB
another DT
sharp JJ
dive NN
if IN
trade NN
figures NNS
for IN
September NNP
, ,
due JJ
for IN
release NN
tomorrow NN
...
```

To match the formatting of the dataset to the `gensim` model, we'll have to do some preprocessing. For example, the model replaces digits with '#' characters, combines separate words into entities where appropriate (e.g., considering "New_York" as a single token instead of two separate words), and utilizes underscores where the raw data uses dashes. We preprocess the dataset to conform to this model schema with the following code (analogous code is used to process the training data):

```
with open("/path/to/pos.train.txt") as f:
        train_dataset_raw = f.readlines()
        train_dataset_raw = [e.split() for e in
        train_dataset_raw if len(e.split()) > 0]

    counter = 0
    while counter < len(train_dataset_raw):
        pair = train_dataset_raw[counter]
        if counter < len(train_dataset_raw) - 1:
            next_pair = train_dataset_raw[counter + 1]
            if (pair[0] + "_" + next_pair[0] in model) and
               (pair[1] == next_pair[1]):
                train_dataset.append([pair[0] + "_" +
                  next_pair[0], pair[1]])
                counter += 2
                continue

        word = re.sub("\d", "#", pair[0])
        word = re.sub("-", "_", word)

        if word in model:
            train_dataset.append([word, pair[1]])
            counter += 1
```

```
            continue

        if "_" in word:
            subwords = word.split("_")
            for subword in subwords:
                if not (subword.isspace() or len(subword) == 0):
                    train_dataset.append([subword, pair[1]])
            counter += 1
            continue

        train_dataset.append([word, pair[1]])
        counter += 1

    with open('/path/to/pos.train.processed.txt', 'w')
    as train_file: for item in train_dataset:
            train_file.write("%s\n" % (item[0] + " " +
                                       item[1]))
```

Now that we've appropriately processed the datasets for use, we can load the words in LevelDB. If the word or phrase is present in the `gensim` model, we can cache that in the LevelDB instance. If not, we randomly select a vector to represent to the token, and cache it so that we remember to use the same vector in case we encounter it again:

```
db = leveldb.LevelDB("data/word2vecdb")
counter = 0
    for pair in train_dataset + test_dataset:
        dataset_vocab[pair[0]] = 1
        if pair[1] not in tags_to_index:
            tags_to_index[pair[1]] = counter
            index_to_tags[counter] = pair[1]
            counter += 1

    nonmodel_cache = {}

    counter = 1
    total = len(dataset_vocab.keys())
    for word in dataset_vocab:
        if counter % 100 == 0:
            print "Inserted %d words out of %d total" % (
              counter, total)
        if word in model:
            db.Put(word, model[word])
        elif word in nonmodel_cache:
            db.Put(word, nonmodel_cache[word])
        else:
            print word
            nonmodel_cache[word] = np.random.uniform(-0.25,
               0.25, 300).
                                astype(np.float32)
            db.Put(word, nonmodel_cache[word])
        counter += 1
```

After running the script for the first time, we can just load our data straight from the database if it already exists:

```
db = leveldb.LevelDB("data/word2vecdb")

with open("data/pos_data/pos.train.processed.txt") as f:
    train_dataset = f.readlines()
    train_dataset = [element.split() for element in
                     train_dataset if
                     len(element.split()) > 0]

with open("data/pos_data/pos.train.processed.txt") as f:
    test_dataset = f.readlines()
    test_dataset = [element.split() for element in test_dataset
                    if len(element.split()) > 0]

counter = 0
for pair in train_dataset + test_dataset:
    dataset_vocab[pair[0]] = 1
    if pair[1] not in tags_to_index:
        tags_to_index[pair[1]] = counter
        index_to_tags[counter] = pair[1]
        counter += 1
```

Finally, we build dataset objects for both training and test datasets, which we can utilize to generate minibatches for training and testing purposes. Building the dataset object requires access to the LevelDB `db`, the `dataset`, a dictionary `tags_to_index` that maps POS tags to indices in the output vector, and a boolean flat `get_all` that determines whether getting the minibatch should retrieve the full set by default:

```
class POSDataset():
    def __init__(self, db, dataset, tags_to_index,
                 get_all=False):
        self.db = db
        self.inputs = []
        self.tags = []
        self.ptr = 0
        self.n = 0
        self.get_all = get_all

        for pair in dataset:
            self.inputs.append(np.fromstring(db.Get(pair[0]),
                               dtype=np.float32))
            self.tags.append(tags_to_index[pair[1]])

        self.inputs = np.array(self.inputs, dtype=np.float32)
        self.tags = np.eye(len(tags_to_index.keys()))
                               [self.tags]

    def prepare_n_gram(self, n):
        self.n = n
```

```
    def minibatch(self, size):
        batch_inputs = []
        batch_tags = []
        if self.get_all:
            counter = 0
            while counter < len(self.inputs) - self.n + 1:
                batch_inputs.append(self.inputs[
                counter:counter+self.n].flatten())
                batch_tags.append(self.tags[counter +
                self.n - 1])
                counter += 1
        elif self.ptr + size < len(self.inputs) - self.n:
            counter = self.ptr
            while counter < self.ptr + size:
                batch_inputs.append(self.inputs
                [counter:counter+self.n].flatten())
                batch_tags.append(self.tags[counter +
                self.n - 1])
                counter += 1
        else:
            counter = self.ptr
            while counter < len(self.inputs) - self.n + 1:
                batch_inputs.append(self.inputs[
                counter:counter+self.n].flatten())
                batch_tags.append(self.tags[counter +
                self.n - 1])
                counter += 1

            counter2 = 0
            while counter2 < size - counter + self.ptr:
                batch_inputs.append(self.inputs[
                  counter2:counter2+self.n].flatten())
                batch_tags.append(self.tags[
     counter2 + self.n - 1])
                counter2 += 1

        self.ptr = (self.ptr + size) % (len(self.inputs) -
                                        self.n)
        return np.array(batch_inputs, dtype=np.float32),
           np.array
                        (batch_tags)



train = POSDataset(db, train_dataset, tags_to_index)
test = POSDataset(db, test_dataset, tags_to_index,
                  get_all=True)
```

Finally, we design our feed-forward network similarly to our approaches in previous chapters. We omit a discussion of the code and refer to the file `feedforward_pos.py` in the book's companion repository. To run the model with 3-gram input vectors, we run the following command:

```
$ python feedforward_pos.py 3

LOADING PRETRAINED WORD2VEC MODEL...
Using a 3-gram model
Epoch: 0001 cost = 3.149141798
Validation Error: 0.336273431778
Then        ``
the        DT
woman        NN
,        RP
after        UH
grabbing        VBG
her        PRP
umbrella        NN
,        RP
went        UH
to        TO
the        PDT
bank        NN
to        TO
deposit        PDT
her        PRP
cash        NN
.        SYM


Epoch: 0002 cost = 2.971566474
Validation Error: 0.300647974014
Then        ``
the        DT
woman        NN
,        RP
after        UH
grabbing        RBS
her        PRP$
umbrella        NN
,        RP
went        UH
to        TO
the        PDT
bank        NN
to        TO
deposit        )
her        PRP$
cash        NN
.        SYM

...
```

Every epoch, we manually inspect the model by parsing the sentence: “The woman, after grabbing her umbrella, went to the bank to deposit her cash.” Within 100 epochs of training, the algorithm achieves over 96% accuracy and nearly perfectly parses the

validation sentence (it makes the understandable mistake of confusing the possessive pronoun and personal pronoun tags for the first appearance of the word "her"). We'll conclude this by including the visualizations of our model's performance using TensorBoard in Figure 7-4.

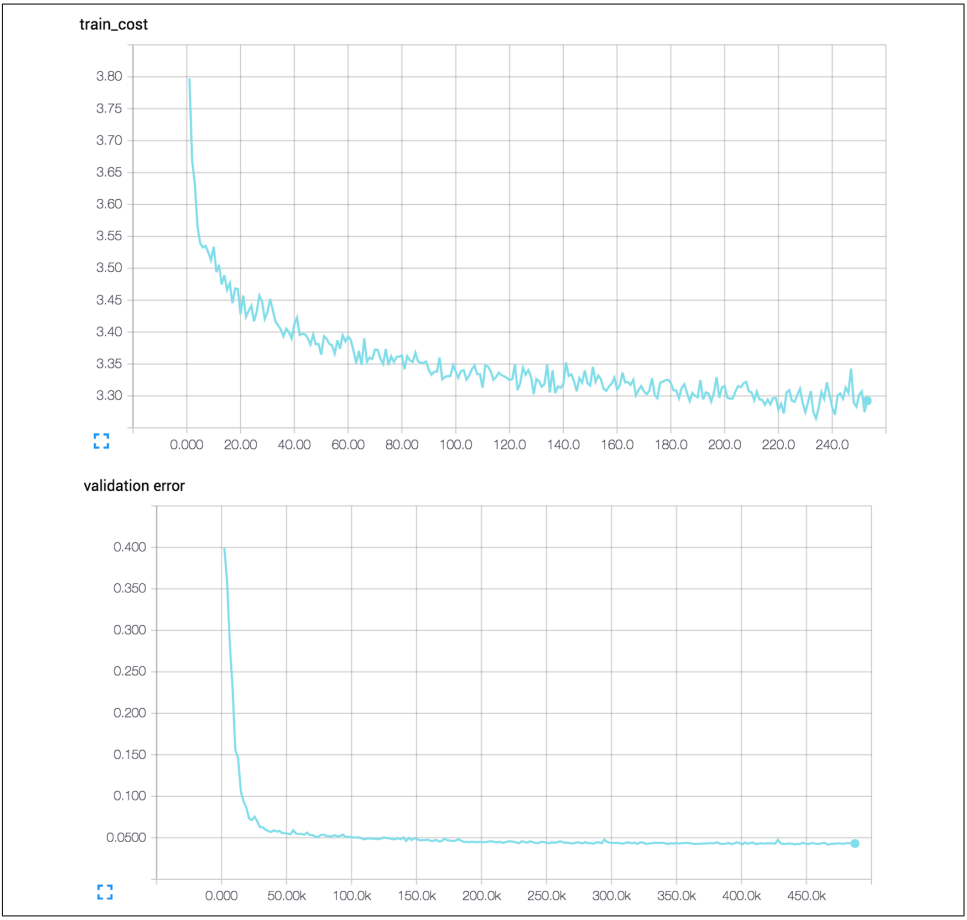


Figure 7-4. TensorBoard visualization of our feedfoward POS tagging model

The POS tagging model was a great exercise, but it was mostly rinsing and repeating concepts we've learned in previous chapters. In the rest of the chapter, we'll start to think about much more complicated sequence-related learning tasks. To tackle these more difficult problems, we'll need to broach brand-new concepts, develop new architectures, and start to explore the cutting edge of modern deep learning research. We'll start by tackling the problem of dependency parsing next.

Dependency Parsing and SyntaxNet

The framework we used to solve the POS tagging task was rather simple. Sometimes we need to be much more creative about how we tackle seq2seq problems, especially as the complexity of the problem increases. In this section, we'll explore strategies that employ creative data structures to tackle difficult seq2seq problems. As a illustrative example, we'll explore the problem of dependency parsing.

The idea behind building a dependency parse tree is to map the relationships between words in a sentence. Take, for example, the dependency in Figure 7-5. The words "I" and "taxi" are children of the word "took," specifically as the subject and direct object of the verb, respectively.

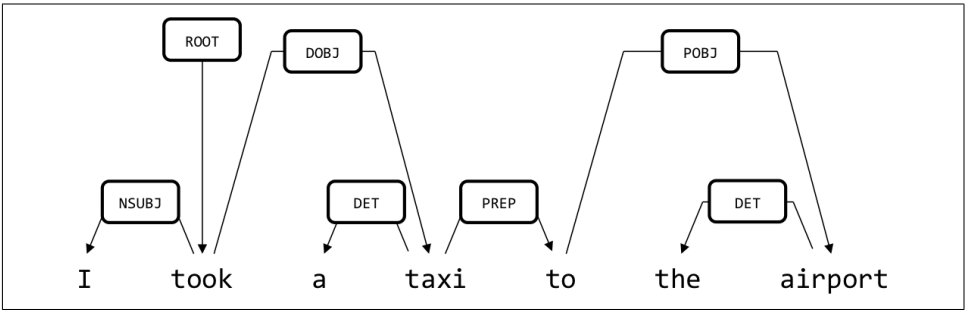


Figure 7-5. An example of a dependency parse, which generates a tree of relationships between words in a sentence

One way to express a tree as a sequence is by linearizing it. Let's consider the examples in Figure 7-6. Essentially, if you have a graph with a root R, and children A (connected by edge r_a), B (connected by edge r_b), and C (connected by edge r_c), we can linearize the representation as (R, r_a, A, r_b, B, r_c, C). We can even represent more complex graphs. Let's assume, for example, that node B actually has two more children named D (connected by edge b_d) and E (connected by edge b_e). We can represent this new graph as (R, r_a, A, r_b, [B, b_d, D, b_e, E], r_c, C).

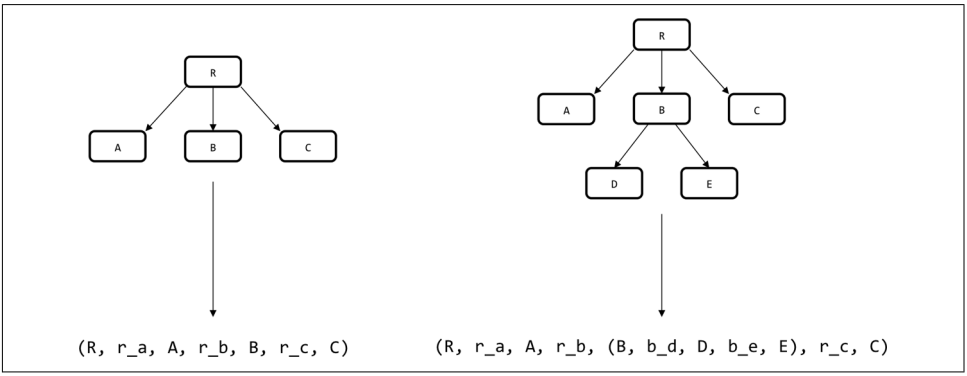


Figure 7-6. We linearize two example trees, the diagrams omit edge labels for the sake of visual clarity

Using this paradigm, we can take our example dependency parse and linearize it, as shown in Figure 7-7.

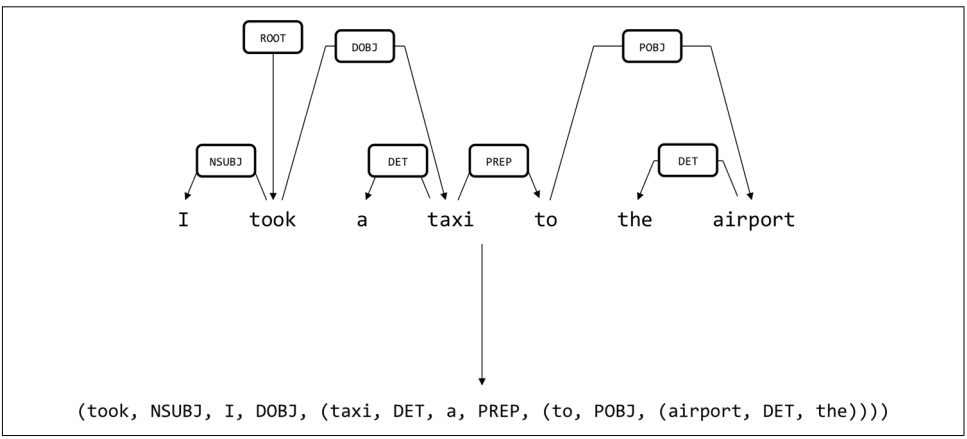


Figure 7-7. Linearization of the dependency parse tree example

One interpretation of of this seq2seq problem would be to read the input sentence and produce a sequence of tokens as an output that represents the linearization of the input's dependency parse. It's not particularly clear, however, how we might port our strategy from the previous section, where there was a clear one-to-one mapping between words and their POS tags. Moreover, we could easily make decisions about a POS tag by looking at the nearby context. For dependency parsing, there's no clear relationship between how words are ordered in the sentence and how tokens in the linearization are ordered. It also seems like dependency parsing tasks us with identifying edges that may span a significantly large number of words. Therefore, at first glance, it seems like this setup directly violates our assumption that we need not take into account any long-term dependencies.

To make the problem more approachable, we instead reconsider the dependency parsing task as finding a sequence of valid "actions" that generates the correct dependency parse. This technique, known as the *arc-standard* system, was first described by Nivre[5] in 2004 and later leveraged in a neural context by Chen and Manning[6] in 2014. In the arc-standard system, we start by putting the first two words of the sentence in the stack and maintaining the remaining words in the buffer, as shown in Figure 7-8.

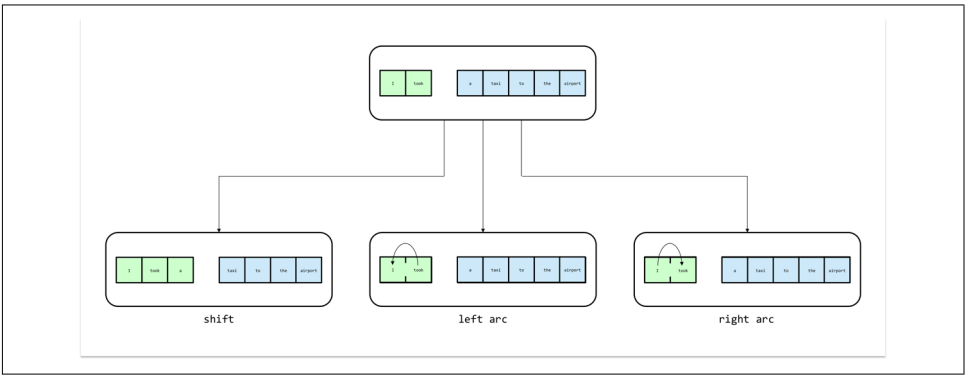


Figure 7-8. At any step, we have three options: to shift a word from the buffer (blue) to the stack (green), to draw an arc from the right element to the left element (left arc), or to draw an arc from the left element to the right element (right arc)

At any step, we can take one of three possible classes of actions:

SHIFT
: Move a word from the buffer to the front of the stack.

LEFT ARC
: Combine the two elements at the front of the stack into a single unit where the root of the rightmost element is the parent node and the root of leftmost element is the child node.

RIGHT ARC
: Combine the two elements at the front of the stack into a single unit where the root of the left element is the parent node and the root of right element is the child node.

5 Nivre, Joakim. "Incrementality in Deterministic Dependency Parsing." Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together. Association for Computational Linguistics, 2004.

6 Chen, Danqi, and Christopher D. Manning. "A Fast and Accurate Dependency Parser Using Neural Networks." EMNLP. 2014.

We note that while there is only one way to perform a SHIFT, the ARC actions can be of many flavors, each differentiated by the dependency label assigned to the arc that is generated. That being said, we'll simplify our discussions and illustrations in this section by considering each decision as a choice among three actions (rather than tens of actions).

We finally terminate this process when the buffer is empty and the stack has one element in it (which represents the full dependency parse). To illustrate this process in its entirety, we illustrate a sequence of actions that generates the dependency parse for our example input sentence in Figure 7-9.

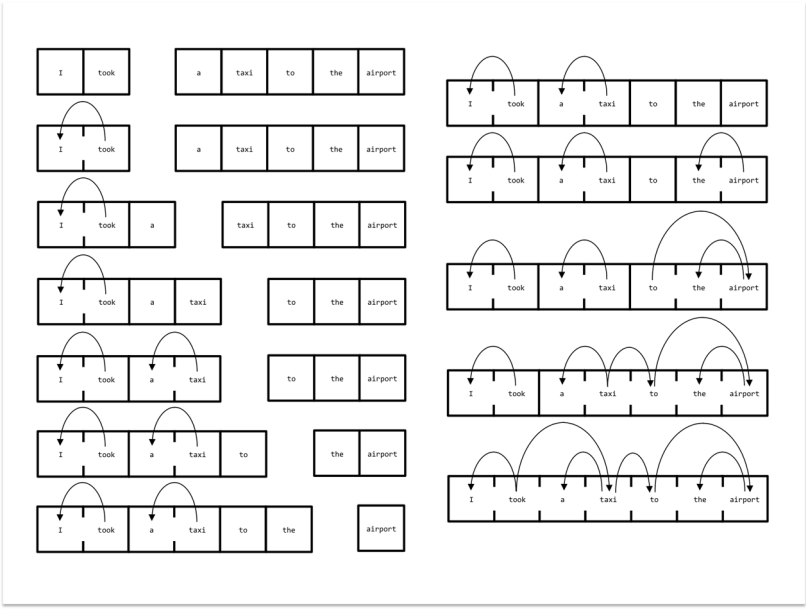


Figure 7-9. A sequence of actions that results in the correct dependency parse; we omit labels

It's not too difficult to reformulate this decision-making framework as a learning problem. At every step, we take the current configuration, we vectorize the configuration by extracting a large number of features that describe the configuration (words in specific locations of the stack/buffer, specific children of the words in these locations, part of speech tags, etc.). During train time, we can feed this vector into a feed-forward network and compare its prediction of the next action to take to a gold standard decision made by a human linguist. To use this model in the wild, we can take the action that the network recommends, apply it to the configuration, and use this new configuration as the starting point for the next step (feature extraction, action prediction, and action application). This process is shown in Figure 7-10.

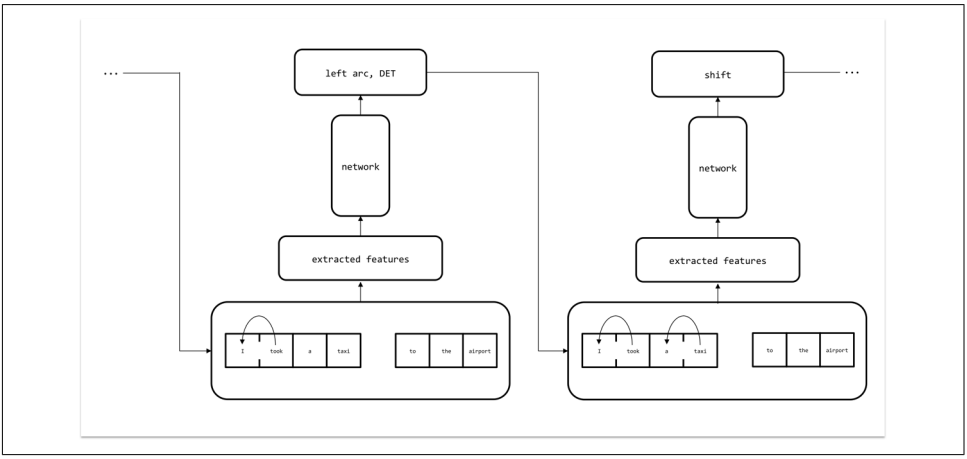


Figure 7-10. A neural framework for arc-standard dependency parsing

Taken together, these ideas form the core for Google's SyntaxNet, the state-of-the-art open source implementation for dependency parsing. Delving into the nitty-gritty aspects of implementation is beyond the scope of this text, but we refer the inspired reader to the open source repository[7], which contains an implementation of Parsey McParseface, the most accurate publicly reported English language parser as of the publication of this text.

Beam Search and Global Normalization

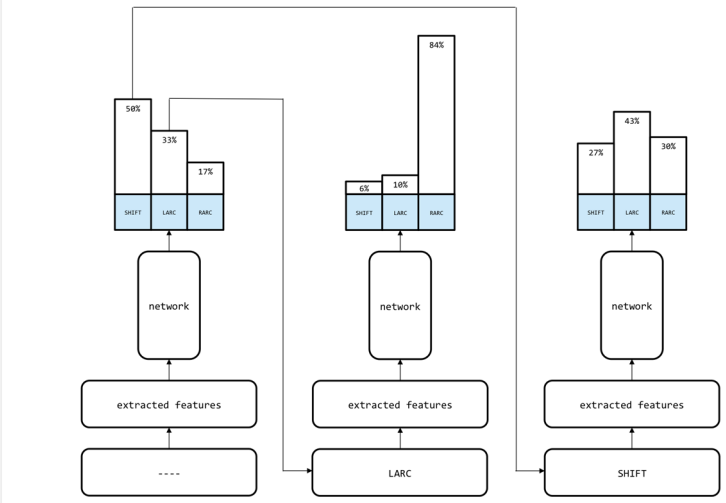
In the previous section, we described naive strategy for deploying SyntaxNet in practice. The strategy was purely *greedy*; that is, we selected prediction with the highest probability without being concerned that we might potentially paint ourselves into a corner by making an early mistake. In the POS example, making an incorrect prediction was largely inconsequential. This is because each prediction could be considered a purely independent subproblem (the results of a given prediction do not affect the inputs of the next step).

This assumption no longer holds in SyntaxNet, because our prediction at step n affects the input we use at step $n + 1$. This implies that any mistake we make will influence all later decisions. Moreover, there's no good way of "going backward" and fixing mistakes when they become apparent. *Garden path sentences* are an extreme case of where this is important. Consider the following sentence: "The complex houses married and single soldiers and their families." The first glance pass-through is confusing. Most people interpret "complex" as an adjective, "houses" as a

7 *https://github.com/tensorflow/models/tree/master/syntaxnet*

noun, and "married" as a past tense verb. This makes little semantic sense though, and starts to break down as the rest of the sentence is read. Instead, we realize that "complex" is a noun (as in a military complex) and that "houses" is a verb. In other words, the sentence implies that the military complex contains soldiers (who may be single or married) and their families. A *greedy* version of SyntaxNet would fail to correct the early parse mistake of considering "complex" as an adjective describing the "houses," and therefore fail on the full version of the sentence.

To remedy this shortcoming, we utilize a strategy known as *beam search*, illustrated in Figure 7-11. We generally leverage beam searches in situations like SyntaxNet, where the output of our network at a particular step influences the inputs used in future steps. The basic idea behind beam search is that instead of greedily selecting the most probable prediction at each step, we maintain a *beam* of the most likely hypothesis (up to a fixed *beam size b*) for the sequence of the first *k* actions and their associated probabilities. Beam searching can be broken up into two major phases: expansion and pruning.



INITIAL		EXPAND		PRUNE		EXPAND		PRUNE	
----	100%	SHIFT	50%	SHIFT	50%	SHIFT,SHIFT	13.5%	LARC,RARC	27.7%
		LARC	33%	LARC	33%	SHIFT,LARC	21.5%	SHIFT,LARC	21.5%
		RARC	17%			SHIFT,RARC	15%		
						LARC,SHIFT	2%		
						LARC,LARC	3.3%		
						LARC,RARC	27.7%		

Figure 7-11. An illustration of using beam search (with beam size 2) while deploying a trained SyntaxNet model

During the *expansion* step, we take each hypothesis and consider it as a possible input to SyntaxNet. Assume SyntaxNet produces a probability distribution over a space of $|A|$ total actions. We then compute the probability of each of the $b|A|$ possible hypotheses for the sequence of the first $k + 1$ actions. Then, during the *pruning* step, we keep only the b hypothesis out of the $b|A|$ total options with the largest probabilities. As Figure 7-11 illustrates, beam searching enables SyntaxNet to correct incorrect predictions post facto by entertaining less probable hypotheses early that might turn out to be more fruitful later in the sentence. In fact, digging deeper into the illustrated example, a greedy approach would have suggested that the correct sequence of moves would have been a SHIFT followed by a LEFT ARC. In reality, the best (highest probability) option would have been to use a LEFT ARC followed by a RIGHT ARC. Beam searching with beam size 2 surfaces this result.

The full open source version takes this a full step further and attempts to bring the concept of beam searching to the process of training the network. As Andor et al. describe in 2016,[8] this process of *global normalization* provides both strong theoretical guarantees and clear performance gains relative to *local normalization* in practice. In a locally normalized network, our network is tasked with selecting the best action given a configuration. The network outputs a score that is normalized using a softmax layer. This is meant to model a probability distribution over all possible actions, provided the actions performed thus far. Our loss function attempts to force the probability distribution to the ideal output (i.e., probability 1 for the correct action and 0 for all other actions). The cross-entropy loss does a spectacular job of ensuring this for us.

In a globally normalized network, our interpretation of the scores is slightly different. Instead of putting the scores through a softmax to generate a per-action probability distribution, we instead add up all the scores for a hypothesis action sequence. One way of ensuring that we select the correct hypothesis sequence is by computing this sum over all possible hypotheses and then applying a softmax layer to generate a probability distribution. We could theoretically use the same cross-entropy loss function as we used in the locally normalized network. The problem with this strategy, however, is that there is an intractably large number of possible hypothesis sequences. Even considering an average sentence length of 10 and a conservative total number of 15 possible actions—1 shift and 7 labels for each of the left and right arcs—this corresponds to 1,000,000,000,000,000 possible hypotheses.

To make this problem tractable, as shown in Figure 7-12, we apply a beam search, with a fixed beam size, until we either 1) reach the end of the sentence, or 2) the correct sequence of actions is no longer contained on the beam. We then construct a loss function that tries to push the "gold standard" action sequence (highlighted in blue) as high as possible on the beam by maximizing its score relative to the other hypotheses. While we won't dive into the details of how we might construct this loss function here, we refer the curious reader to the original paper by Andor et al. in 2016.[9] The paper also describes a more sophisticated POS tagger that uses global normalization and beam search to significantly increase accuracy (compared to the POS tagger we built earlier in the chapter).

8 Andor, Daniel, et al. "Globally Normalized Transition-Based Neural Networks." arXiv preprint arXiv: 1603.06042 (2016).

9 Andor, Daniel et al. "Globally Normalized Transition-Based Neural Networks." arXiv preprint arXiv: 1603.06042 (2016).

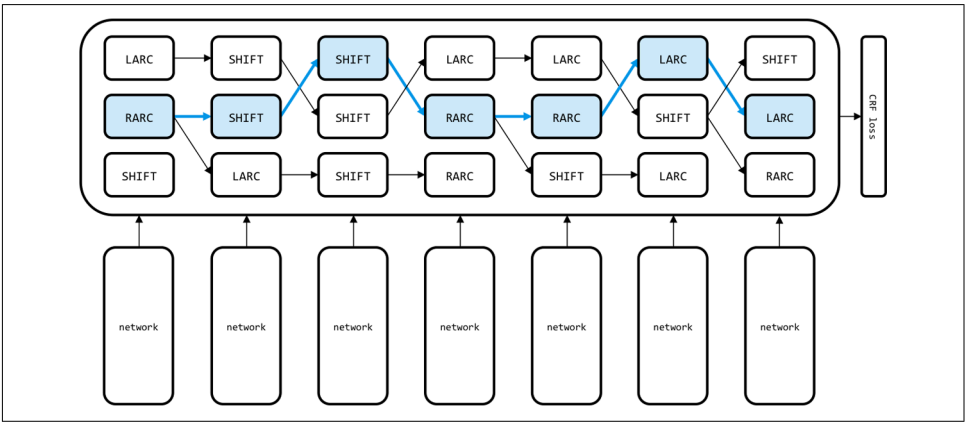


Figure 7-12. We can make global normalization in SyntaxNet tractable by coupling training and beam search

A Case for Stateful Deep Learning Models

While we've explored several tricks to adapt feed-forward networks to sequence analysis, we've yet to truly find an elegant solution to sequence analysis. In the POS tagger example, we made the explicit assumption that we can ignore long-term dependencies. We were able to overcome some of the limitations of this assumption by introducing the concepts of beam searching and global normalization, but even still, the problem space was constrained to situations in which there was a one-to-one mapping between elements in the input sequence to elements in the output sequence. For example, even in the dependency parsing model, we had to reformulate the problem to discover a one-to-one mapping between a sequence of input configurations while constructing the parse tree and arc-standard actions.

Sometimes, however, the task is far more complicated than finding a one-to-one mapping between input and output sequences. For example, we might want to develop a model that can consume an entire input sequence at once and then conclude if the sentiment of the entire input was positive or negative. We'll build a simple model to perform this task later in the chapter. We may want an algorithm that consumes a complex input (such as an image) and generate a sentence, one word at a time, describing the input. We may event want to translate sentences from one language to another (e.g., from English to French). In all of these instances, there's no obvious mapping between input tokens and output tokens. Instead, the process is more like the situation in Figure 7-13.

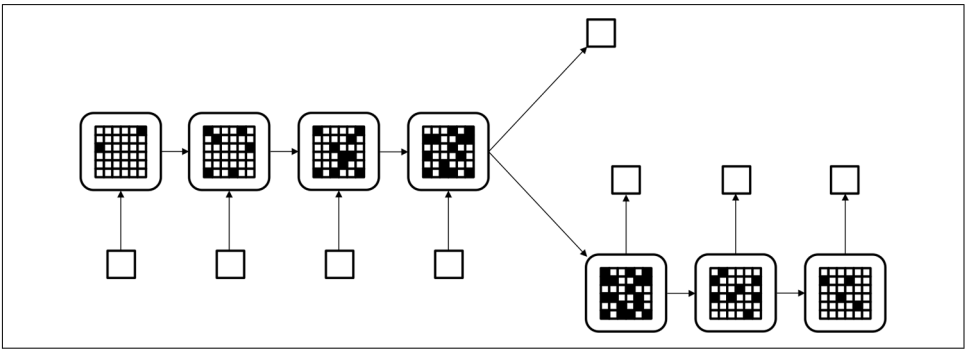


Figure 7-13. The ideal model for sequence analysis can store information in memory over long periods of time, leading to a coherent "thought" vector that it can use to generate an answer

The idea is simple. We want our model to maintain some sort of memory over the span of reading the input sequence. As it reads the input, the model should able to modify this memory bank, taking into account the information that it observes. By the time it has reached the end of the input sequence, the internal memory contains a "thought" that represents the key pieces of information, that is, the meaning, of the original input. We should then, as shown in Figure 7-13, be able to use this thought vector to either produce a label for the original sequence or produce an appropriate output sequence (translation, description, abstractive summary, etc.).

The concept here isn't something we've explored in any of the previous chapters. Feed-forward networks are inherently "stateless." After it's been trained, the feed-forward network is a static structure. It isn't able to maintain memories between inputs, or change how it processes an input based on inputs it has seen in the past. To execute this strategy, we'll need to reconsider how we construct neural networks to create deep learning models that are "stateful." To do this, we'll have to return to how we think about networks on an individual neuron level. In the next section, we'll explore how *recurrent connections* (as opposed to the feed-forward connections we have studied this far) enable models to maintain state as we describe a class of models known as *recurrent neural networks* (RNNs).

Recurrent Neural Networks

RNNs were sfirst introduced in the 1980s, but have regained popularity recently due to several intellectual and hardware breakthroughs that have made them tractable to train. RNNs are different from feed-forward networks because they leverage a special type of neural layer, known as recurrent layers, that enable the network to maintain state between uses of the network.

Figure 7-14 illustrates the neural architecture of a recurrent layer. All of the neurons have both 1) incoming connections emanating from all of the neurons of the previous layer and 2) outgoing connections leading to all of the neurons to the subsequent layer. We notice here, however, that these aren't the only connections that neurons of a recurrent layer have. Unlike a feed-forward layer, recurrent layers also have recurrent connections, which propagate information between neurons of the same layer. A fully connected recurrent layer has information flow from every neuron to every other neuron in its layer (including itself). Thus a recurrent layer with r neurons has a total of r^2 recurrent connections.

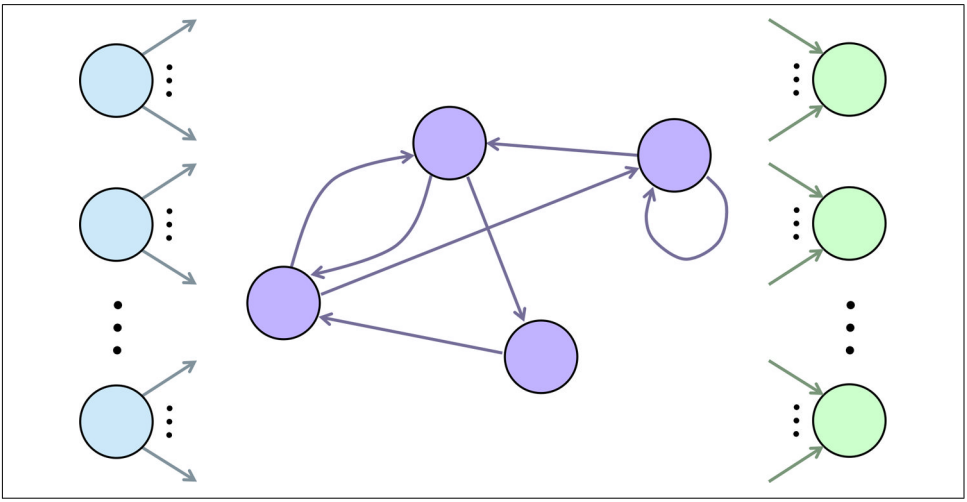


Figure 7-14. A recurrent layer contains recurrent connections, that is to say, connections between neurons that are located in the same layer

To better understand how RNNs work, let's explore how one functions after it's been appropriately trained. Every time we want to process a new sequence, we create a fresh instance of our model. We can reason about networks that contain recurrent layers by dividing the lifetime of the network instance into discrete time steps. At each time step, we feed the model the next element of the input. Feedforward connections represent information flow from one neuron to another where the data being transferred is the computed neuronal activation from the current time step. Recurrent connections, however, represent information flow where the data is the stored neuronal activation from the *previous* time step. Thus, the activations of the neurons in a recurrent network represent the accumulating state of the network instance. The initial activations of neurons in the recurrent layer are parameters of our model, and we determine the optimal values for them just like we determine the optimal values for the weights of each connection during the process of training.

It turns out that, given a fixed lifetime (say t time steps) of an RNN instance, we can actually express the instance as a feed-forward network (albeit irregularly structured). This clever transformation, illustrated in Figure 7-15, is often referred to as "unrolling" the RNN through time. Let's consider the example RNN in the figure. We'd like to map a sequence of two inputs (each dimension 1) to a single output (also of dimension 1). We perform the transformation by taking the neurons of the single recurrent layer and replicating them it t times, once for each time step. We similarly replicate the neurons of the input and output layers. We redraw the feed-forward connections within each time replica just as they were in the original network. Then we draw the recurrent connections as feed-forward connections from each time replica to the next (since the recurrent connections carry the neuronal activation from the previous time step).

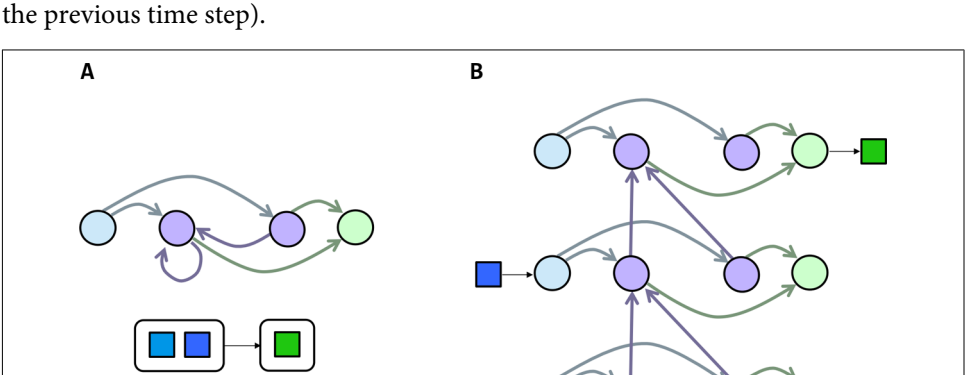


Figure 7-15. We can run an RNN through time to express it as a feedforward network that we can train using backpropagation

We can also now train the RNN by computing the gradient based on the unrolled version. This means that all of the backpropagation techniques that we utilized for feed-forward networks also apply to training RNNs. We do run into one issue, however. After every batch of training examples we use, we need to modify the weights based on the error derivatives we calculate. In our unrolled network, we have sets of connections that all correspond to the same connection in the original RNN. The error derivatives calculated for these unrolled connections, however, are not guaranteed to be (and, in practice, probably won't be) equal. We can circumvent this issue by averaging or summing the error derivatives over all the connections that belong to the same set. This allows us to utilize an error derivative that considers all of the dynamics acting on the weight of a connection as we attempt to force the network to construct an accurate output.

The Challenges with Vanishing Gradients

Our motivation for using a stateful network model hinges on this idea of capturing long-term dependencies in the input sequence. It seems reasonable that an RNN with a large memory bank (i.e., a significantly sized recurrent layer) would be able to summarize these dependencies. In fact, from a theoretical perspective, Kilian and Siegelmann demonstrated in 1996 that the RNN is a universal functional representation.[10] In other words, with enough neurons and the right parameter settings, an RNN can be used to represent any functional mapping between input and output sequences.

The theory is promising, but it doesn't necessarily translate to practice. While it is nice to know that it is *possible* for an RNN to represent any arbitrary function, it is more useful to know whether it is *practical* to teach the RNN a realistic functional mapping from scratch by applying gradient descent algorithms. If it turns out to be impractical, we'll be in hot water, so it will be useful for us to be rigorous in exploring this question. Let's start our investigation by considering the simplest possible RNN, shown in Figure 7-16, with a single input neuron, a single output neuron, and a fully connected recurrent layer with one neuron.

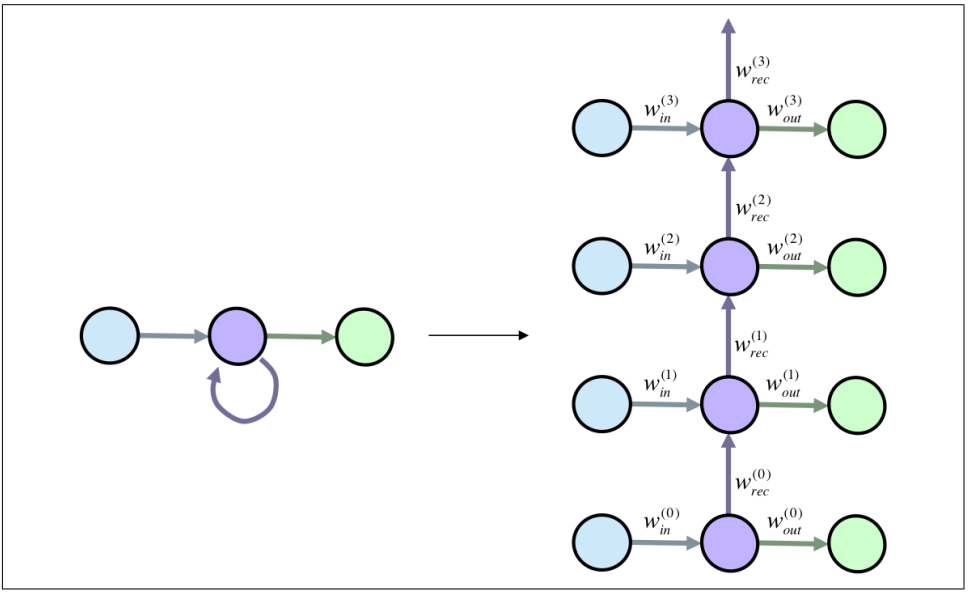


Figure 7-16. A single neuron, fully connected recurrent layer (both compressed and unrolled) for the sake of investigating gradient-based learning algorithms

10 Kilian, Joe, and Hava T. Siegelmann. "The dynamic universality of sigmoidal neural networks." Information and computation 128.1 (1996): 48-56.

Let's start off simple. Given nonlinearity f, we can express the activation $h^{(t)}$ of the the hidden neuron of the recurrent layer at time step t as follows, where $i^{(t)}$ is the incoming logit from the input neuron at time step t:

$$h^{(t)} = f\left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right)$$

Let's try to compute how the activation of the hidden neuron changes in response to changes to the input logit from k time steps in the past. In analyzing this component of the backpropagation gradient expressions, we can start to quantify how much "memory" is retained from past inputs. We start by taking the partial derivative and apply the chain rule:

$$\frac{\partial h^{(t)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right) \frac{\partial}{\partial i^{(t-k)}} \left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right)$$

Because the values of the input and recurrent weights are independent of the input logit at time step $t-k$, we can further simplify this expression:

$$\frac{\partial h^{(t)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right) w_{rec}^{(t-1)} \frac{\partial h^{(t-1)}}{\partial i^{(t-k)}}$$

Because we care about the magnitude of this derivative, we can take the absolute value of both sides. We also know that for all common nonlinearities (the tanh, logistic, and ReLU nonlinearities), the maximum value of $|f'|$ is at most 1. This leads to the following recursive inequality:

$$\left|\frac{\partial h^{(t)}}{\partial i^{(t-k)}}\right| \leq \left|w_{rec}^{(t-1)}\right| \cdot \left|\frac{\partial h^{(t-1)}}{\partial i^{(t-k)}}\right|$$

We can continue to expand this inequality recursively until we reach the base case, at step $t-k$:

$$\left|\frac{\partial h^{(t)}}{\partial i^{(t-k)}}\right| \leq \left|w_{rec}^{(t-1)}\right| \cdot \ldots \cdot \left|w_{rec}^{(t-k)}\right| \cdot \left|\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}}\right|$$

We can evaluate this partial derivative similarly to how we proceeded previously:

$$h^{(t-k)} = f\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right)$$

$$\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right) \frac{\partial}{\partial i^{(t-k)}} \left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right)$$

In this expression, the hidden activation at time $t-k-1$ is independent of the value of the input at $t-k$. Thus we can rewrite this expression as:

$$\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right) w_{in}^{(t-k)}$$

Finally, taking the absolute value on both sides and again applying the observation about the maximum value of $|f'|$, we can write:

$$\left|\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}}\right| \leq \left|w_{in}^{(t-k)}\right|$$

This results in the final inequality (which we can simplify because we constrain the connections at different time steps to have equal value):

$$\left|\frac{\partial h^{(t)}}{\partial i^{(t-k)}}\right| \leq \left|w_{rec}^{(t-1)}\right| \cdot \ldots \cdot \left|w_{rec}^{(t-k)}\right| \cdot \left|w_{in}^{(t-k)}\right| = \left|w_{rec}\right|^{k} \cdot w_{in}$$

This relationship places a strong upper bound on how much a change in the input at time $t - k$ can impact the hidden state at time t. Because the weights of our model are initialized to small values at the beginning of training, the value of this derivative approaches zero as k increases. In other words, the gradient quickly diminishes when it's computed with respect to inputs several time steps into the past, severely limiting our model's ability to learn long-term dependencies. This issue is commonly referred to as the problem of *vanishing gradients,* and it severely impacts the learning capabilities of vanilla recurrent neural networks. In order to address this limitation, we will spend the next section exploring an extraordinarily influential twist on recurrent layers known as long short-term memory.

Long Short-Term Memory (LSTM) Units

In order to combat the problem of vanishing gradients, Sepp Hochreiter and Jürgen Schmidhuber introduced the *long short-term memory* (LSTM) architecture. The basic principle behind the architecture was that the network would be designed for the purpose of reliably transmitting important information many time steps into the future. The design considerations resulted in the architecture shown in Figure 7-17.

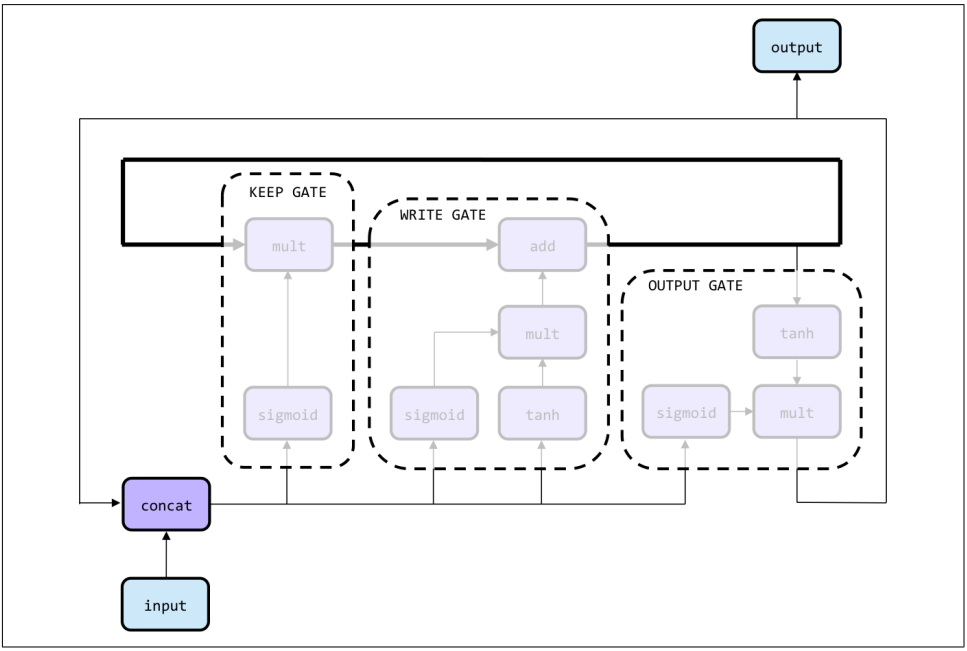


Figure 7-17. The architecture of an LSTM unit, illustrated at a tensor (designated by arrows) and operation (designated by the purple blocks) level

For the purposes of this discussion, we'll take a step back from the individual neuron level and start talking about the network as collection tensors and operations on tensors. As the figure indicates, the LSTM unit is composed of several key components. One of the core components of the LSTM architecture is the *memory cell*, a tensor represented by the bolded loop in the center of the figure. The memory cell holds critical information that it has learned over time, and the network is designed to effectively maintain useful information in the memory cell over many time steps. At every time step, the LSTM unit modifies the memory cell with new information with three different phases. First, the unit must determine how much of the previous memory to keep. This is determined by the *keep gate*, shown in detail in Figure 7-18.

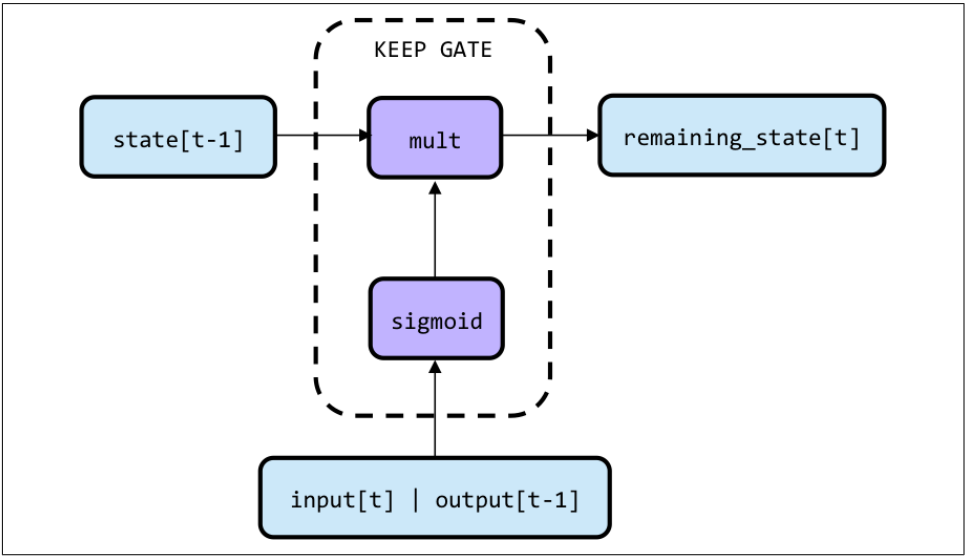


Figure 7-18. Architecture of the keep gate of an LSTM unit

The basic idea of the keep gate is simple. The memory state tensor from the previous time step is rich with information, but some of that information may be stale (and therefore might need to be erased). We figure out which elements in the memory state tensor are still relevant and which elements are irrelevant by trying to compute a bit tensor (a tensor of zeros and ones) that we multiply with the previous state. If a particular location in the bit tensor holds a 1, it means that location in the memory cell is still relevant and ought to be kept. If that particular location instead held a 0, it means that the location in the memory cell is no longer relevant and ought to be eased. We approximate this bit tensor by concatenating the input of this time step and the LSTM unit's output from the previous time step and applying a sigmoid layer to the resulting tensor. A sigmoidal neuron, as you may recall, outputs a value that is either very close to 0 or very close to 1 most of the time (the only exception is when the input is close to zero). As a result, the output of the sigmoidal layer is a close approximation of a bit tensor, and we can use this to complete the keep gate.

Once we've figured out what information to keep in the old state and what to erase, we're ready to think about what information we'd like to write into the memory state. This part of the LSTM unit is known as the write gate, and it's depicted in Figure 7-19. This is broken down into two major parts. The first component is figuring out what information we'd like to write into the state. This is computed by the tanh layer to create an intermediate tensor. The second component is figuring out which components of this computed tensor we actually want to include into the new state and which we want to toss before writing. We do this by approximating a bit vector of 0's and 1's using the same strategy (a sigmoidal layer) as we used in the keep

gate. We multiply the bit vector with our intermediate tensor and then add the result to create the new state vector for the LSTM.

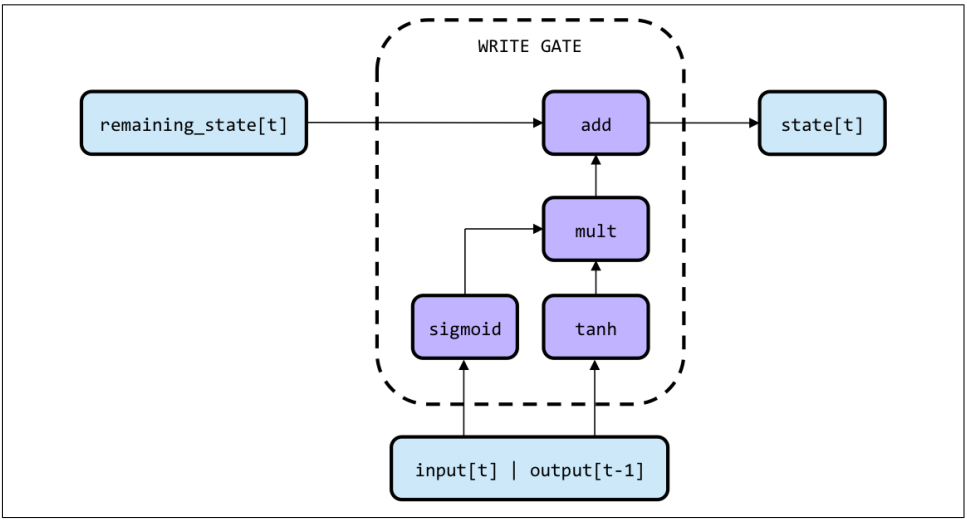


Figure 7-19. Architecture of the write gate of an LSTM unit

Finally, at every time step, we'd like the LSTM unit to provide an output. While we could treat the state vector as the output directly, the LSTM unit is engineered to provide more flexibility by emitting an output tensor that is a "interpretation" or external "communication" of what the state vector represents. The architecture of the output gate is shown in Figure 7-20. We use a nearly identical structure as the write gate: 1) the tanh layer creates an intermediate tensor from the state vector, 2) the sigmoid layer produces a bit tensor mask using the current input and previous output, and 3) the intermediate tensor is multiplied with the bit tensor to produce the final output.

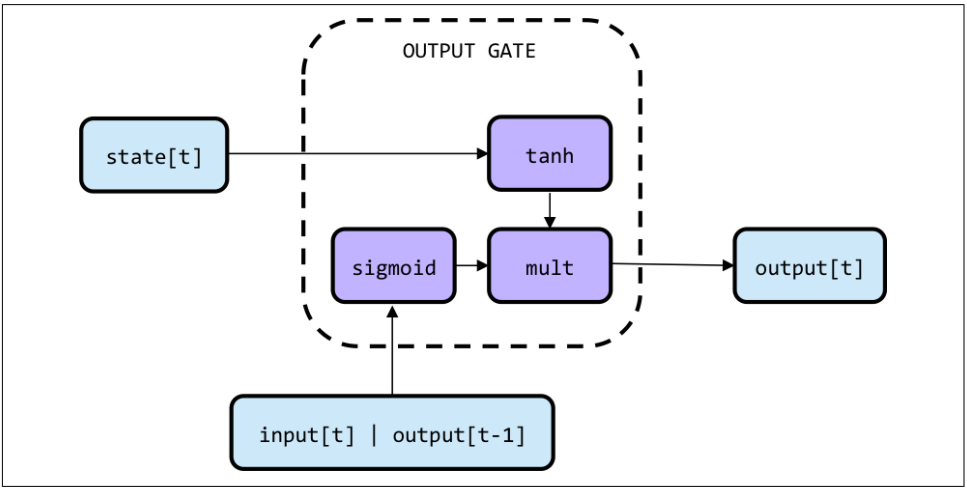


Figure 7-20. Architecture of the output gate of an LSTM unit

So why is this better than using a raw RNN unit? The key observation is how information propagates through the network when we unroll the LSTM unit through time. The unrolled architecture is shown in Figure 7-21. At the very top, we can observed the propagation of the state vector, whose interactions are primarily linear through time. The result is that the gradient that relates an input several time steps in the past to the current output does not attenuate as dramatically as in the vanilla RNN architecture. This means that the LSTM can learn long-term relationships much more effectively than our original formulation of the RNN.

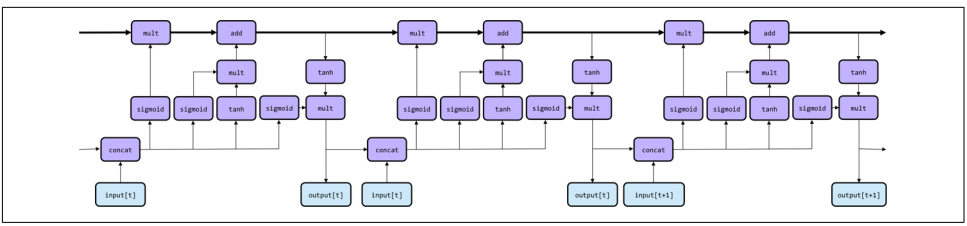


Figure 7-21. Unrolling an LSTM unit through time

Finally, we want to understand how easy it is to generate arbitrary architectures with LSTM units. How "composable" are LSTMs? Do we need to sacrifice any flexibility to use LSTM units instead of a vanilla RNN? Well, just as we can we can stack RNN layers to create more expressive models with more capacity, we can similarly stack LSTM units, where the input of the second unit is the output of the first unit, the input of the third unit is the output of the second, and so on. An illustration of how this works is shown in Figure 7-22, with a multicellular made of two LSTM units. This means that anywhere we use a vanilla RNN layer, we can easily substitute an LSTM unit.

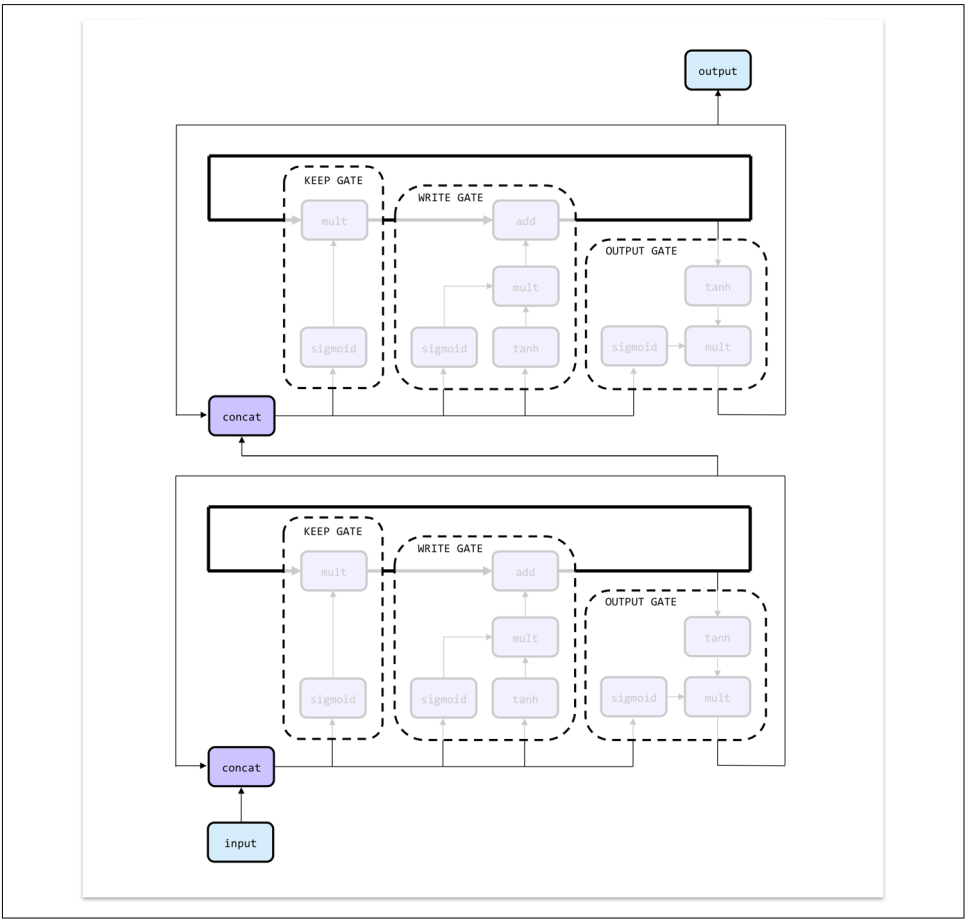


Figure 7-22. Composing LSTM units as one might stack recurrent layers in a neural network

Now that we have overcome the issue of vanishing gradients and understand the inner workings of LSTM units, we're ready to dive into the implementation of our first RNN models.

TensorFlow Primitives for RNN Models

There are several primitives that TensorFlow provides that we can use out of the box in order to build RNN models. First, we have `tf.RNNCell` objects that represent either an RNN layer or an LSTM unit:

```
cell_1 = tf.nn.rnn_cell.BasicRNNCell(num_units, input_size=None,
                                     activation=tanh)
cell_2 = tf.nn.rnn_cell.BasicLSTMCell(num_units,
```

```
                                        forget_bias=1.0,
                                        input_size=None,
                                        state_is_tuple=True,
                                        activation=tanh)
    cell_3 = tf.nn.rnn_cell.LSTMCell(num_units, input_size=None,
                                     use_peepholes=False,
                                     cell_clip=None,
                                     initializer=None,
                                     num_proj=None,
                                     proj_clip=None,
                                     num_unit_shards=1,
                                     num_proj_shards=1,
                                     forget_bias=1.0,
                                     state_is_tuple=True,
                                     activation=tanh)
    cell_4 = tf.nn.rnn_cell.GRUCell(num_units, input_size=None,
                                    activation=tanh)
```

The `BasicRNNCell` abstraction represents a vanilla recurrent neuron layer. The `BasicLSTMCell` represents a simple implementation of the LSTM unit, and the `LSTMCell` represents an implementation with more configuration options (peephole structures, clipping of state values, etc.). The TensorFlow library also includes a variation of the LSTM unit known as the *Gated Recurrent Unit*, proposed in 2014 by Yoshua Bengio's group. The critical initialization variable for all of these cells is the size of the hidden state vector or `num_units`.

In addition to the primitives, there are several wrappers to add to our arsenal. If we want to stack recurrent units or layers, we can use the following:

```
cell_1 = tf.nn.rnn_cell.BasicLSTMCell(10)
cell_2 = tf.nn.rnn_cell.BasicLSTMCell(10)
full_cell = tf.nn.rnn_cell.MultiRNNCell([cell_1, cell_2])
```

We can also use a wrapper to apply dropout to the inputs and outputs of an LSTM with specified input and output keep probabilities:

```
cell_1 = tf.nn.rnn_cell.BasicLSTMCell(10)
tf.nn.rnn_cell.DropoutWrapper(cell_1, input_keep_prob=1.0,
                              output_keep_prob=1.0,
                              seed=None)
```

Finally, we complete the RNN by wrapping everything into the appropriate TensorFlow RNN primitive:

```
outputs, state = tf.nn.dynamic_rnn(cell, inputs,
                                   sequence_length=None,
                                   initial_state=None,
                                   dtype=None,
                                   parallel_iterations=None,
                                   swap_memory=False,
                                   time_major=False,
                                   scope=None)
```

The `cell` is the `RNNCell` object we've compiled thus far. If `time_major == False` (which is the default setting), `inputs` must be a tensor of the shape `[batch_size, max_time, ...]`. Otherwise if `time_major == True`, we must have `inputs` with the shape: `[max_time, batch_size, ...]`. We refer the curious reader to the TensorFlow documentation for elucidation of the other configuration parameters.

The result of calling `tf.nn.dynamic_rnn` is a tensor representing the outputs of the RNN along with the final state vector. If `time_major == False`, then `outputs` will be of shape `[batch_size, max_time, cell.output_size]`. Otherwise, `outputs` will have shape `[max_time, batch_size, cell.output_size]`. We can expect `state` to be of size `[batch_size, cell.state_size]`.

Now that we have an understanding of the tools at our disposal in constructing recurrent neural networks in TensorFlow, we'll build our first LSTM in the next section, focused on the task of sentiment analysis.

Implementing a Sentiment Analysis Model

In this section, we attempt to analyze the sentiment of movie reviews taken from the Large Movie Review Dataset. This dataset consists of 50,000 reviews from IMDB, each of which labeled as having positive or negative sentiment. We use a simple LSTM model leveraging dropout to learn how to classify the sentiment of movie reviews. The LSTM model will consume the movie review one word at a time. Once it has consumed the entire review, we'll use its output as the basis of a binary classification to map the sentiment to be "positive" or "negative." Let's start off by loading the dataset. To do this, we'll utilize the helper library `tflearn`. We can install `tflearn` by running the following command:

```
$ pip install tflearn
```

Once we've installed the package, we can download the dataset, prune the vocabulary to only include the 30,000 most common words, pad each input sequence up to a length 500 words, and process the labels:

```
from tflearn.data_utils import to_categorical, pad_sequences
from tflearn.datasets import imdb

train, test, _ = imdb.load_data(path='data/imdb.pkl',
                 n_words=30000,
                                 valid_portion=0.1)
trainX, trainY = train
testX, testY = test

trainX = pad_sequences(trainX, maxlen=500, value=0.)
testX = pad_sequences(testX, maxlen=500, value=0.)
```

```
trainY = to_categorical(trainY, nb_classes=2)
testY = to_categorical(testY, nb_classes=2)
```

The inputs here are now 500-dimensional vectors. Each vector corresponds to a movie review where the i^{th} component of the vector corresponds to the index of the i^{th} word of the review in our global dictionary of 30,000 words. To complete the data preparation, we create a special Python class designed to serve minibatches of a desired size from the underlying dataset:

```
class IMDBDataset():
    def __init__(self, X, Y):
        self.num_examples = len(X)
        self.inputs = X
        self.tags = Y
        self.ptr = 0


    def minibatch(self, size):
        ret = None
        if self.ptr + size < len(self.inputs):
            ret =  self.inputs[self.ptr:self.ptr+size],
                   self.tags[self.ptr:self.ptr+size]
        else:
            ret = np.concatenate((self.inputs[self.ptr:],
                                  self.inputs[:size-len(
                                  self.inputs[self.ptr:])])),
                  np.concatenate((self.tags[self.ptr:],
                                  self.tags[:size-len(
                                  self.tags[self.ptr:])]))
        self.ptr = (self.ptr + size) % len(self.inputs)

        return ret

train = IMDBDataset(trainX, trainY)
val = IMDBDataset(testX, testY)
```

We use the `IMDBDataset` Python class to serve both the training and validation sets we'll use while training our sentiment analysis model.

Now that the data is ready to go, we'll begin to construct the sentiment analysis model, step by step. First, we'll want to map each word in the input review to a word vector. To do this, we'll utilize an embedding layer, which, as you may recall from the last chapter, is a simple lookup table that stores an embedding vector that corresponds to each word. Unlike in previous examples, where we treated the learning of the word embeddings as a separate problem (i.e., by building a Skip-Gram model), we'll learn the word embeddings jointly with the sentiment analysis problem by treating the embedding matrix as a matrix of parameters in the full problem. We accomplish this by using the TensorFlow primitives for managing embeddings (remember that `input` represents one full minibatch at a time, not just one movie review vector):

```
def embedding_layer(input, weight_shape):
    weight_init = tf.random_normal_initializer(stddev=(
                    1.0/weight_shape[0])**0.5)
    E = tf.get_variable("E", weight_shape,
                        initializer=weight_init)
    incoming = tf.cast(input, tf.int32)
    embeddings = tf.nn.embedding_lookup(E, incoming)
    return embeddings
```

We then take the result of the embedding layer and build an LSTM with dropout using the primitives we saw in the previous section. We do some extra work to pull out the last output emitted by the LSTM using the `tf.slice` and `tf.squeeze` operators, which find the exact slice that contains the last output of the LSTM and then eliminates the unnecessary dimension. The change in dimensions is as follows: `[batch_size, max_time, cell.output_size]` to `[batch_size, 1, cell.output_size]` to `[batch_size, cell.output_size]`.

The implementation of the LSTM can be achieved as follows:

```
def lstm(input, hidden_dim, keep_prob, phase_train):
        lstm = tf.nn.rnn_cell.BasicLSTMCell(hidden_dim)
        dropout_lstm = tf.nn.rnn_cell.DropoutWrapper(lstm,
                        input_keep_prob=keep_prob,
                           output_keep_prob=keep_prob)
        # stacked_lstm = tf.nn.rnn_cell.MultiRNNCell(
                         [dropout_lstm] * 2,
                         state_is_tuple=True)
        lstm_outputs, state = tf.nn.dynamic_rnn(dropout_lstm,
                              input, dtype=tf.float32)
        return tf.squeeze(tf.slice(lstm_outputs,
                           [0, tf.shape(
                            lstm_outputs)[1]-1, 0],
                                 [tf.shape(lstm_outputs)[0],
                                 1, tf.shape(
                                 lstm_outputs)[2]]))
```

We top it all off using a batch-normalized hidden layer, identical to the ones we've used time and time again in previous examples. Stringing all of these components together, we can build the inference graph:

```
def inference(input, phase_train):
    embedding = embedding_layer(input, [30000, 512])
    lstm_output = lstm(embedding, 512, 0.5, phase_train)
    output = layer(lstm_output, [512, 2], [2], phase_train)
    return output
```

We omit the other boilerplate involved in setting up summary statistics, saving intermediate snapshots, and creating the session because it's identical to the other models we have built in this book; we refer the reader to the source code in the GitHub repository. We can then run and visualize the performance of our model using TensorBoard (Figure 7-23).

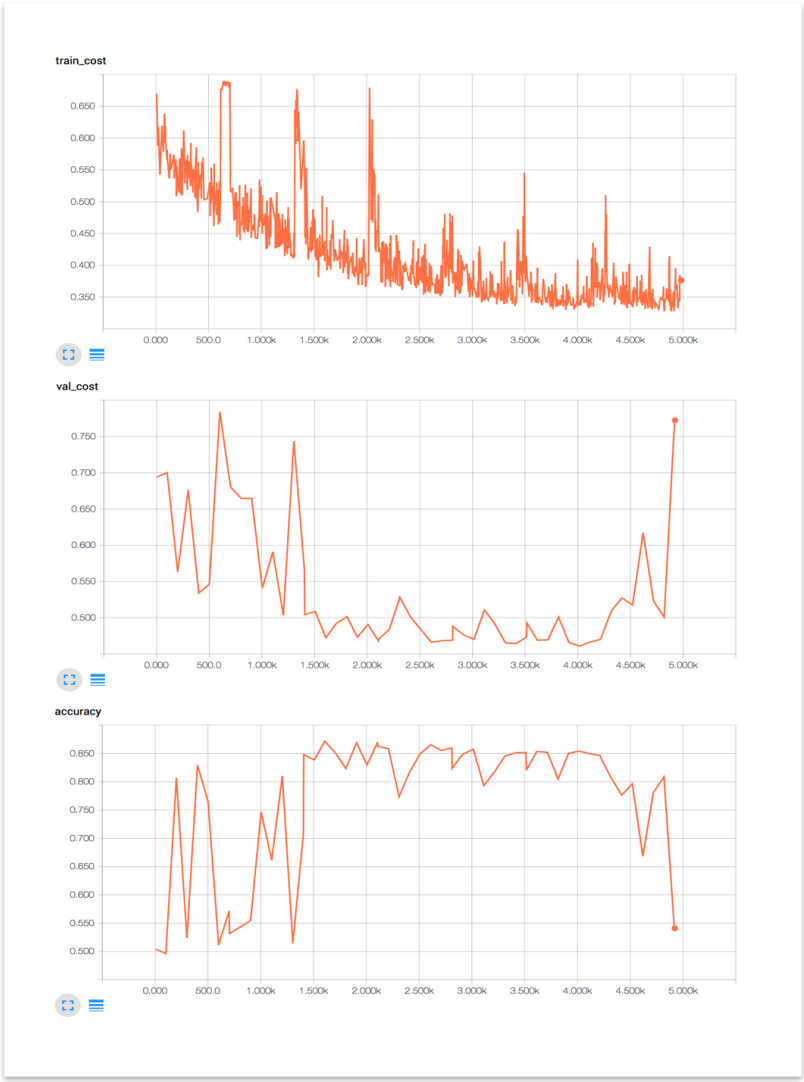


Figure 7-23. Training cost, validation cost, and accuracy of our movie review sentiment model

At the beginning of training, the model struggles slightly with stability, and toward the end of the training, the model clearly starts to overfit as training cost and validation cost significantly diverge. At its optimal performance, however, the model performs rather effectively and generalizes to approximately 86% accuracy on the test set. Congratulations! You've built your first recurrent neural network.

Solving seq2seq Tasks with Recurrent Neural Networks

Now that we've built a strong understanding of recurrent neural networks, we're ready to revisit the problem of seq2seq. We started off this chapter with an example of a seq2seq task: mapping a sequence of words in a sentence to a sequence of POS tags. Tackling this problem was tractable because we didn't need to take into account long-term dependencies to generate the appropriate tags. But there are several seq2seq problems, such as translating between languages or creating a summary for a video, where long-term dependencies are crucial to the to the success of the model. This is where RNNs come in.

The RNN approach to seq2seq looks a lot like the autoencoder we discussed in the previous chapter. The seq2seq model is composed of two separate networks. The first network is known as the *encoder* network. The encoder network is a recurrent network (usually one that uses LSTM units) that consumes the entire input sequence. The goal of the encoder network is to generate a condensed understanding of the input and summarize it into a singular thought represented by the final state of the encoder network. Then we use a *decoder* network, whose starting state is initialized with the final state of the encoder network, to produce the target output sequence token by token. At each step, the decoder network consumes its own output from the previous time step as the current time step's input. The entire process is visualized in Figure 7-24.

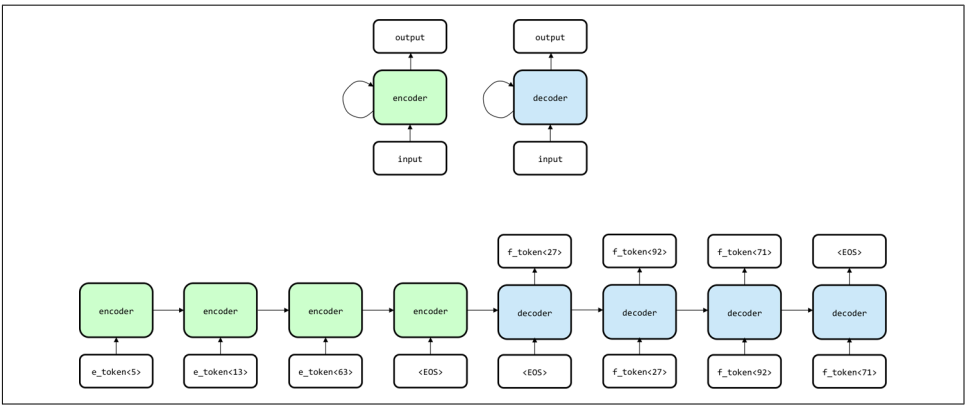


Figure 7-24. Illustration of how we use an encoder/decoder recurrent network schema to tackle seq2seq problems

In this this setup, we are attempting to translate an American sentence into French. We tokenize the input sentence and use an embedding (similarly to our approach in the sentiment analysis model we built in the previous section), one word at a time as an input to the encoder network. At the end of the sentence, we use a special "end of sentence" (EOS) token to indicate the end of the input sequence to the encoder net-

work. Then we take the hidden state of the encoder network and use that as the initialization of the decoder network. The first input to the decoder network is the EOS token, and the output is interpreted as the first word of the predicted French translation. From that point onward, we use the output of the decoder network as the input to itself at the next time step. We continue until the decoder network emits an EOS token as its output, at which point we know that the network has completed producing the translation of the original English sentence. We'll dissect practical, open source implementation of this network (with a couple of enhancements and tricks to improve accuracy) later in this chapter.

The seq2seq RNN architecture can also be reappropriated for the purpose of learning good embeddings of sequences. For example, Kiros et al. in 2015 invented the notion of a *skip-thought vector*,[11] which borrowed architectural characteristics from both the autoencoder framework and Skip-Gram model discussed in Chapter 6. The skip-thought vector was generated by dividing up a passage into a set of triplets consisting of consecutive sentences. The authors utilized a single encoder network and two decoder networks, as shown in Figure 7-25.

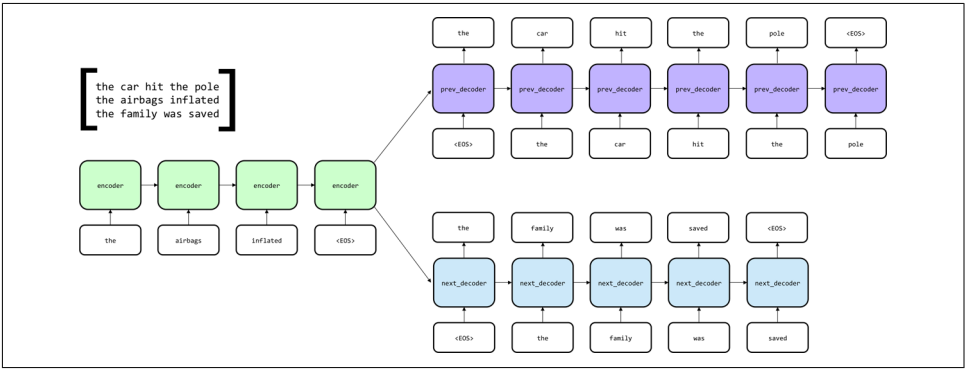


Figure 7-25. The skip-thought seq2seq architecture to generate embedding representations of entire sentences

The encoder network consumed the sentence for which we wanted to generate a condensed representation (which was stored in the final hidden state of the encoder network). Then came the decoding step. The first of the decoder networks would take that representation as the initialization of its own hidden state and attempt to reconstruct the sentence that appeared prior to the input sentence. The second decoder network would instead attempt the sentence that appeared immediately after the input sentence. The full system was trained end to end on these triplets, and once completed, could be utilized to generate seemingly cohesive passages of text in addi-

11 Kiros, Ryan, et al. "Skip-Thought Vectors." Advances in neural information processing systems. 2015.

tion to improve performance on key sentence-level classification tasks. Here's an example of story generation, excerpted from the original paper:

```
she grabbed my hand .
"come on . "
she fluttered her back in the air .
"i think we're at your place . I ca n't come get you . "
he locked himself back up
" no . she will . "
kyrian shook his head
```

Now that we've developed an understanding of how to leverage recurrent neural networks to tackle seq2seq problems, we're almost ready to try to build our own. Before we get there, however, we've got one more major challenge to tackle, and we'll address it head-on in the next section when we discuss the concept of attentions in seq2seq RNNs.

Augmenting Recurrent Networks with Attention

Let's think harder about the translation problem. If you've ever attempted to learn a foreign language, you'll know that there are several things that are helpful when trying to complete a translation. First it's helpful to read the full sentence to understand the concept you would like to convey. Then you write out the translation one word at a time, each word following logically from the word you wrote previously. But one important aspect of translation is that as you compose the new sentence, you often refer back to the original text, focusing on specific parts that are relevant to your current translation. At each step, you are paying attention to the most relevant parts of the original "input" so you can make the best decision about the next word to put on the page.

Let's think back to our approach to seq2seq. By consuming the full input and summarizing it into a "thought" inside its hidden state, the encoder network effectively achieves the first part of the translation process. By using the previous output as its current input, the decoder network achieves the second part of the translation process. This phenomenon of *attention* has yet to be captured by our approach to seq2seq, and this is the final building block we'll need to engineer.

Currently, the sole input to the decoder network at a given time step t is its output at time step $t - 1$. One way to give the decoder network some vision into the original sentence is by giving the decoder access to all of the outputs from the encoder network (which we previously had completely ignored). These outputs are interesting to us because they represent how the encoder network's internal state evolves after seeing each new token. A proposed implementation of this strategy is shown in Figure 7-26.

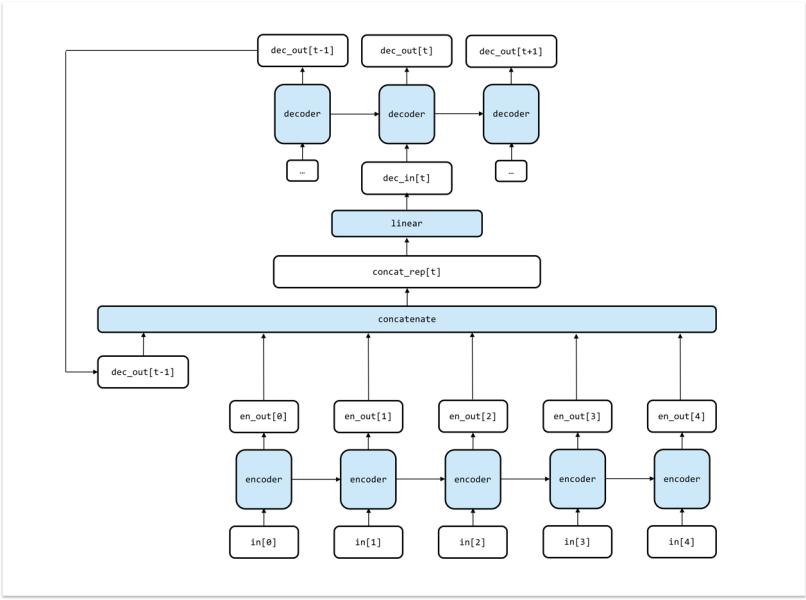


Figure 7-26. An attempt at engineering attentional abilities in a seq2seq architecture. This attempt falls short because it fails to dynamically select the most relevant parts of the input to focus on.

This approach has a critical flaw, however. The problem here is that at every time step, the decoder considers all of the outputs of the encoder network in the exact same way. However, this is clearly not the case for a human during the translation process. We focus on different aspects of the original text when working on different parts of the translation. The key realization here is that it's not enough to merely give the decoder access to all the outputs. Instead, we must engineer a mechanism by which the decoder network can dynamically pay attention to a specific subset of the encoder's outputs.

We can fix this problem by changing the inputs to the concatenation operation, using the proposal in Bahdanau et al. 2015 as inspiration.[12] Instead of directly using the raw outputs from the encoder network, we perform a weighting operation on the encoder's outputs. We leverage the decoder network's state at time $t - 1$ as the basis for the weighting operation.

12 Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." arXiv preprint arXiv:1409.0473 (2014).

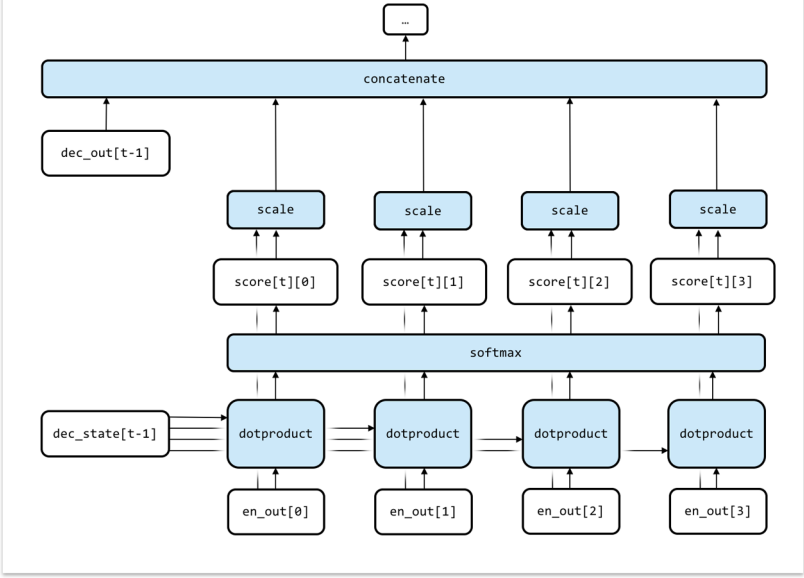


Figure 7-27. A modification to our original proposal that enables a dynamic attentional mechanism based on the hidden state of the decoder network in the previous time step

The weighting operation is illustrated in Figure 7-27. First we create a scalar (a single number, not a tensor) relevance score for each of the encoder's outputs. The score is generated by computing the dot product between each encoder output and the decoder's state at time $t - 1$. We then normalize these scores using a softmax operation. Finally, we use these normalized scores to individually scale the encoder's outputs before plugging them into the concatenation operation. The key here is that the relative scores computed for each encoder output signify how important that particular encoder output is to the decision for the decoder at time step t. In fact, as we'll see later, we can visualize which parts of the input are most relevant to the translation at each time step by inspecting the output of the softmax!

Armed with this strategy for engineering attention into seq2seq architectures, we're finally ready to get our hands dirty with an RNN model for translating English sentences into French. But before we jump in, it's worth noting that attentions are incredibly applicable in problems that extend beyond language translation. Attentions can be important in speech-to-text problems, where the algorithm learns to dynamically pay attention to corresponding parts of the audio while transcribing the audio into text. Similarly, attentions can be used to improve image captioning algorithms by helping the captioning algorithm focus on specific parts of the input image while writing out the caption. Anytime there are particular parts of the input that are

highly correlated to correctly producing corresponding segments of the output, attentions can dramatically improve performance.

Dissecting a Neural Translation Network

State-of-the-art neural translation networks use a number of different techniques and advancements that build on the basic seq2seq encoder-decoder architecture. Attention, as detailed in the previous section, is an important and critical architectural improvement. In this section, we will dissect a fully implemented neural machine translation system, complete with the data processing steps, building the model, training it, and eventually using it as a translation system to convert English phrases to French phrases! We'll pursue this exploration by working with a simplified version of the official TensorFlow machine translation tutorial code.[13]

The pipeline used in training and eventually using a neural machine translation system is very similar to that of most machine learning pipelines: gather data, prepare the data, construct the model, train the model, evaluate the model's progress, and eventually use the trained model to predict or infer something useful. We review each of these steps here.

We first gather the data from the WMT'15 repository, which houses large corpora used in training translation systems. For our use case, we'll be using the English-to-French data. Note that if we want to be able to translate to or from different languages, we would have to train a model from scratch with the new data. We then preprocess our data into a format that is easily usable by our models during training and inference time. This will involve some amount of cleaning and tokenizing the sentences in each of the English and French phrases. What follows now is a set of techniques used in preparing the data, and later we will present the implementations of the techniques.

The first step is to parse sentences and phrases into formats that are more compatible with the model by *tokenization*. This is the process by which we discretize a particular English or French sentence into its constituent tokens. For instance, a simple word-level tokenizer will consume the sentence "I read." to produce the array ["I", "read", "."], or it would consume the French sentence "Je lis." to produce the array ["Je", "lis", "."]. A character-level tokenizer may break the sentence into individual characters or into pairs of characters like ["I", " ", "r", "e", "a", "d", "."] and ["I ", "re", "ad", "."], respectively. One kind of tokenization may work better than the other, and each has its pros and cons. For instance, a word-level tokenizer will ensure that the model produces words that are from some dictionary, but the size of the dictionary may be too large to efficiently choose from during decoding. This is in fact a known issue and some-

13 This code can be found at: *https://github.com/tensorflow/tensorflow/tree/r0.7/tensorflow/models/rnn/translate*.

thing that we'll address in the coming discussions. On the other hand, the decoder using a character-level tokenization may not produce intelligible outputs, but the total dictionary that the decoder must choose from is much smaller, as it is simply the set of all printable ASCII characters. In this tutorial, we use a word-level tokenization, but we encourage the reader to experiment with different tokenizations to observe the effects this has. It is worth noting that we must also add a special EOS, or end-of-sequence character, to the end of all output sequences because we need to provide a definitive way for the decoder to indicate that it has reached the end of its decoding. We can't use regular punctuation because we cannot assume that we are translating full sentences. Note that we do not need EOS characters in our source sequences because we are feeding these in pre-formatted and do not need an end-of-sequence character for ourselves to denote the end of our source sequence.

The next optimization involves further modifying how we represent each source and target sequence, and we introduce a concept called *bucketing*. This is a method employed primarily in sequence-to-sequence tasks, especially machine translation, that helps the model efficiently handle sentences or phrases of different lengths. We first describe the naive method of feeding in training data and illustrate the shortcomings of this approach. Normally, when feeding in encoder and decoder tokens, the length of the source sequence and the target sequence is not always equal between pairs of examples. For example, the source sequence may have length X, and the target sequence may have length Y. It may seem that we need different seq2seq networks to accommodate each (X, Y) pair, yet this immediately seems wasteful and inefficient. Instead, we can do a little better if we *pad* each sequence up to a certain length, as shown in Figure 7-28, assuming we use a word-level tokenization and that we've appended EOS tokens to our target sequences.

I	read	.	<PAD>	<PAD>	<PAD>	<PAD>
Je	lis	.	<EOS>	<PAD>	<PAD>	<PAD>
See	you	in	a	little	while	.
A	tout	a	l'heure	<EOS>	<PAD>	<PAD>
			...			

Figure 7-28. Naive strategy for padding sequences

This step saves us the trouble of having to construct a different seq2seq model for each pair of source and target lengths. However, this introduces a different issue: if there were a very long sequence, it would mean that we would have to pad every other sequence *up to that length*. This would make a short sequence padded to the end take as much computational resources as a long one with few PAD tokens, which is wasteful and could introduce a major performance hit to our model. We could consider breaking up every sentence in the corpus into phrases such that the length of

each phrase does not exceed a certain maximum limit, but it's not clear how to break the corresponding translations. This is where bucketing helps us.

Bucketing is the idea that we can place encoder and decoder pairs into buckets of similar size, and only pad up to the maximum length of sequences in each respective bucket. For instance, we can denote a set of buckets, [(5, 10), (10, 15), (20, 25), (30, 40)], where each tuple in the list is the maximum length of the source sequence and target sequence, respectively. Borrowing the preceding example, we can place the pair of sequences (["I", "read", "."], ["Je", "lis", ".", "EOS"]) in the first bucket, as the source sequence is smaller than 5 tokens and the target sequence is smaller than 10 tokens. We would then place the (["See", "you", "in", "a", "little", "while"], ["A", "tout", "a", "l'heure", "EOS"]) in the second bucket, and so on. This technique allows us to compromise between the two extremes, where we only need to pad as much as necessary, as shown in Figure 7-29.

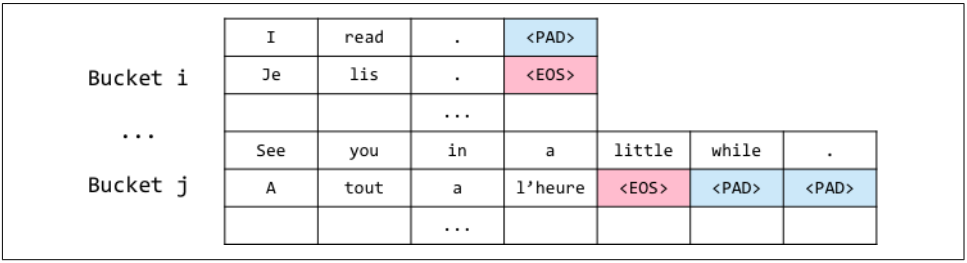


Figure 7-29. Padding sequences with buckets

Using bucketing shows a considerable speedup during training and test time, and allows developers and frameworks to write very optimized code to leverage the fact that any sequence from a bucket will have the same size and pack the data together in ways that allow even further GPU efficiency.

With the sequences properly padded, we need to add one additional token to the target sequences: *a GO token*. This GO token will signal to the decoder that decoding needs to begin, at which point it will take over and begin decoding.

The last improvement we make in the data preparation side is that we reverse the source sequences. Researchers found that doing so improved performance, and this has become a standard trick to try when training neural machine translation models. This is a bit of an engineering hack, but consider the fact that our fixed-size neural state can only hold so much information, and information encoded while processing the beginning of the sentence may be overwritten while encoding later parts of the sentence. In many language pairs, the beginning of sentences is harder to translate than the end of sentences, so this hack of reversing the sentence improves translation accuracy by giving the beginning of the sentence the last say on what final state is encoded. With these ideas in place, the final sequences look as they do in Figure 7-30.

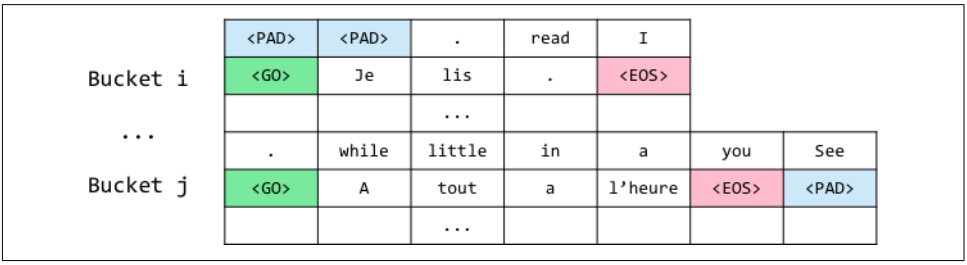


Figure 7-30. Final padding scheme with buckets, reversing the inputs, and adding the GO token

With these techniques described, we can now detail the implementation. The ideas are in a method called `get_batch()` in the code. This method collects a single batch of training data, given the `bucket_id`, which is chosen from the training loop, and the data. The result of this method includes the tokens in the source and target sequences and applies all of the techniques we just discussed, including the padding with buckets and reversing the inputs:

```
def get_batch(self, data, bucket_id):
  encoder_size, decoder_size = self.buckets[bucket_id]
  encoder_inputs, decoder_inputs = [], []
```

We first declare placeholders for each of the inputs that the encoder and decoder consume:

```
  for _ in xrange(self.batch_size):
    encoder_input, decoder_input = random.choice(data[
      bucket_id])

    # Encoder inputs are padded and then reversed.
    encoder_pad = [data_utils.PAD_ID] * (encoder_size - len(
                                          encoder_input))
    encoder_inputs.append(list(reversed(encoder_input +
                                        encoder_pad)))

    # Decoder inputs get an extra "GO" symbol,
    # and are then padded.
    decoder_pad_size = decoder_size - len(decoder_input) - 1
    decoder_inputs.append([data_utils.GO_ID] + decoder_input +
                          [data_utils.PAD_ID] *
                             decoder_pad_size)
```

Given the size of the batch, we gather that many encoder and decoder sequences:

```
   # Now we create batch-major vectors from the data selected
   # above.
  batch_encoder_inputs, batch_decoder_inputs, batch_weights =
    [], [], []

  # Batch encoder inputs are just re-indexed encoder_inputs.
```

```
    for length_idx in xrange(encoder_size):
      batch_encoder_inputs.append(
          np.array([encoder_inputs[batch_idx][length_idx]
                    for batch_idx in xrange(self.batch_size)],
                                          dtype=np.int32))

    # Batch decoder inputs are re-indexed decoder_inputs,
    # we create weights.
    for length_idx in xrange(decoder_size):
      batch_decoder_inputs.append(
          np.array([decoder_inputs[batch_idx][length_idx]
                    for batch_idx in xrange(self.batch_size)],
                                          dtype=np.int32))
```

With additional bookkeeping, we make sure that vectors are batch-major, meaning that the batch size is the first dimension in the tensor, and we resize the previously defined placeholders into the correct shape:

```
      # Create target_weights to be 0 for targets that
      # are padding.
      batch_weight = np.ones(self.batch_size, dtype=np.float32)
      for batch_idx in xrange(self.batch_size):
        # We set weight to 0 if the corresponding target is
        # a PAD symbol.
        # The corresponding target is decoder_input shifted
        # by 1 forward.
        if length_idx < decoder_size - 1:
          target = decoder_inputs[batch_idx][length_idx + 1]
        if length_idx == decoder_size - 1 or
              target == data_utils.PAD_ID:
          batch_weight[batch_idx] = 0.0
      batch_weights.append(batch_weight)
    return batch_encoder_inputs, batch_decoder_inputs,
    batch_weights
```

Finally, we set the target weights of zero to those tokens that are simply the PAD token.

With the data preparation now done, we are ready to begin building and training our model! We first detail the code used during training and test time, and abstract the model away for now. When doing so, we can make sure we understand the high-level pipeline, and we will then study the seq2seq model in more depth. As always, the first step during training is to load our data:

```
def train():
 """Train a en->fr translation model using WMT data."""
 # Prepare WMT data.
 print("Preparing WMT data in %s" % FLAGS.data_dir)
 en_train, fr_train, en_dev, fr_dev, _, _ =
   data_utils.prepare_wmt_data(
     FLAGS.data_dir, FLAGS.en_vocab_size, FLAGS.fr_vocab_size)
```

After instantiating our TensorFlow session, we first create our model. Note that this method is flexible to a number of different architectures as long as they respect the input and output requirements detailed by the `train()` method:

```
with tf.Session() as sess:
 # Create model.
 print("Creating %d layers of %d units." % (FLAGS.num_layers,
                                            FLAGS.size))
 model = create_model(sess, False)
```

We now process the data using various utility functions into buckets that are later used by `get_batch()` to fetch the data. We also create an array of real numbers from 0 to 1 that roughly dictate the likelihood of selecting a bucket, normalized by the size of buckets. When `get_batch()` selects buckets, it will do so respecting these probabilities:

```
 # Read data into buckets and compute their sizes.
 print ("Reading development and training data (limit: %d)."
        % FLAGS.max_train_data_size)
 dev_set = read_data(en_dev, fr_dev)
 train_set = read_data(en_train, fr_train,
                       FLAGS.max_train_data_size)
 train_bucket_sizes = [len(train_set[b]) for b in xrange(
                        len(_buckets))]
 train_total_size = float(sum(train_bucket_sizes))

 # A bucket scale is a list of increasing numbers
 # from 0 to 1 that we'll use to select a bucket.
 # Length of [scale[i], scale[i+1]] is proportional to
 # the size if i-th training bucket, as used later.
 train_buckets_scale = [sum(train_bucket_sizes[:i + 1]) /
                            train_total_size
                        for i in xrange(len(
                            train_bucket_sizes))]
```

With data ready, we now enter our main training loop. We initialize various loop variables, like `current_step` and `previous_losses` to 0 or empty. It is important to note that each cycle in the `while` loop denotes one epoch, which is the terminology for looping through one batch of training data. Therefore, per epoch, we select a `bucket_id`, get a batch using `get_batch`, and then *step* forward in our model with the data:

```
# This is the training loop.
step_time, loss = 0.0, 0.0
current_step = 0
previous_losses = []
while True:
  # Choose a bucket according to data distribution.
  # We pick a random number
  # in [0, 1] and use the corresponding interval
  # in train_buckets_scale.
```

```
random_number_01 = np.random.random_sample()
bucket_id = min([i for i in xrange(len(
                train_buckets_scale))
                 if train_buckets_scale[i] >
                   random_number_01])

# Get a batch and make a step.
start_time = time.time()
encoder_inputs, decoder_inputs, target_weights =
   model.get_batch(
    train_set, bucket_id)
_, step_loss, _ = model.step(sess, encoder_inputs,
                             decoder_inputs,
                             target_weights, bucket_id,
                             False)
```

We measure the loss incurred during prediction time as well as keep track of other running metrics:

```
 step_time += (time.time() - start_time) /
   FLAGS.steps_per_checkpoint
loss += step_loss / FLAGS.steps_per_checkpoint
current_step += 1
```

Lastly, every so often, as dictated by a global variable, we will carry out a number of tasks. First, we print statistics for the previous batch, such as the loss, the learning rate, and the perplexity. If we find that the loss is not decreasing, it is possible that the model has fallen into a local optima. To assist the model in escaping this, we anneal the learning rate so that it won't make large leaps in any particular direction. At this point, we also save a copy of the model and its weights and activations to disk:

```
 # Once in a while, we save checkpoint, print statistics,
 # and run evals.
if current_step % FLAGS.steps_per_checkpoint == 0:
  # Print statistics for the previous epoch.
  perplexity = math.exp(float(loss)) if loss <
    300 else float("inf")
  print ("global step %d learning rate %.4f
          step-time %.2f perplexity "
         "%.2f" % (model.global_step.eval(),
                   model.learning_rate.eval(),
                   step_time, perplexity))
  # Decrease learning rate if no improvement was seen over
  # last 3 times.
  if len(previous_losses) > 2 and loss > max(
         previous_losses[-3:]):
    sess.run(model.learning_rate_decay_op)
  previous_losses.append(loss)
  # Save checkpoint and zero timer and loss.
  checkpoint_path = os.path.join(FLAGS.train_dir,
     "translate.ckpt")
  model.saver.save(sess, checkpoint_path,
```

```
                        global_step=model.global_step)
    step_time, loss = 0.0, 0.0
```

Finally, we will measure the model's performance on a held-out development set. By doing so, we can measure the generalization of the model and see if is improving, and if so, at what rate. We again fetch data using `get_batch`, but this time only use `bucket_id` from the held-out set. We again step through the model, but this time without updating any of the weights because the last argument in the `step()` method is `True` as opposed to `False` during the main training loop; we will discuss the semantics of `step()` later. We measure this evaluation loss and display it to the user:

```
 # Run evals on development set and print
 # their perplexity.
for bucket_id in xrange(len(_buckets)):
  if len(dev_set[bucket_id]) == 0:
    print("  eval: empty bucket %d" % (bucket_id))
    continue
  encoder_inputs, decoder_inputs,
  target_weights = model.get_batch(
      dev_set, bucket_id)
  # attns, _, eval_loss, _ = model.step(sess,
    encoder_inputs, decoder_inputs,
  _, eval_loss, _ = model.step(sess, encoder_inputs,
                               decoder_inputs,
                               target_weights,
                               bucket_id,
                               True)
  eval_ppx = math.exp(float(eval_loss)) if eval_loss <
              300 else float(
      "inf")
  print("  eval: bucket %d perplexity %.2f" % (
         bucket_id, eval_ppx))
sys.stdout.flush()
```

We also have another major use case for our model: single-use prediction. In other words, we want to be able to use our trained model to translate new sentences that we, or other users, provide. To do so, we use the `decode()` method. This method will essentially carry out the same functions as was done in the evaluation loop for the held-out development set. However, the largest difference is that during training and evaluation, we never needed the model to translate the output embeddings to output tokens that are human-readable, which is something we do here. We detail this method now.

Because this is a separate mode of computation, we need to again instantiate the TensorFlow session and create the model, or load a saved model from a previous checkpoint step:

```
def decode():
 with tf.Session() as sess:
```

```
# Create model and load parameters.
model = create_model(sess, True)
```

We set the batch size to 1, as we are not processing any new sentences in parallel, and only load the input and output vocabularies, as opposed to the data itself:

```
 model.batch_size = 1  # We decode one sentence at a time.
# Load vocabularies.
en_vocab_path = os.path.join(FLAGS.data_dir,
                             "vocab%d.en" %
                               FLAGS.en_vocab_size)
fr_vocab_path = os.path.join(FLAGS.data_dir,
                             "vocab%d.fr" %
                               FLAGS.fr_vocab_size)
en_vocab, _ = data_utils.initialize_vocabulary(
              en_vocab_path)
_, rev_fr_vocab = data_utils.initialize_vocabulary(
                  fr_vocab_path)
```

We set the input to standard input so that the user can be prompted for a sentence:

```
 # Decode from standard input.
sys.stdout.write("> ")
sys.stdout.flush()
sentence = sys.stdin.readline()
```

While the sentence provided is nonempty, it is tokenized and truncated if it exceeds a certain maximum length:

```
while sentence:
 # Get token-ids for the input sentence.
 token_ids = data_utils.sentence_to_token_ids(
             tf.compat.as_bytes(sentence), en_vocab)
 # Which bucket does it belong to?
 bucket_id = len(_buckets) - 1
 for i, bucket in enumerate(_buckets):
   if bucket[0] >= len(token_ids):
     bucket_id = i
     break
 else:
   logging.warning("Sentence truncated: %s", sentence)
```

While we don't fetch any data, `get_batch()` will now format the data into the right shapes and prepare it for use in `step()`:

```
 # Get a 1-element batch to feed the sentence to
 # the model.
encoder_inputs, decoder_inputs, target_weights =
  model.get_batch(
    {bucket_id: [(token_ids, [])]}, bucket_id)
```

We step through the model, and this time, we want the `output_logits`, or the unnormalized log-probabilities of the output tokens, instead of the loss. We decode this

with an output vocabulary and truncate the decoding at the first EOS token observed. We then print this French sentence or phrase to the user and await the next sentence:

```
 # Get output logits for the sentence.
 _, _, output_logits = model.step(sess, encoder_inputs,
                                  decoder_inputs,
                                  target_weights,
                                  bucket_id, True)
# This is a greedy decoder - outputs are just argmaxes
# of output_logits.
outputs = [int(np.argmax(logit, axis=1))
           for logit in output_logits]
# If there is an EOS symbol in outputs, cut them
# at that point.
if data_utils.EOS_ID in outputs:
  outputs = outputs[:outputs.index(data_utils.EOS_ID)]
# Print out French sentence corresponding to outputs.
print(" ".join([tf.compat.as_str(rev_fr_vocab[output])
                for output in outputs]))
print("> ", end="")
sys.stdout.flush()
sentence = sys.stdin.readline()
```

This concludes the high-level details of training and using the models. We have largely abstracted away the fine details of the model itself, and for some users, this may be sufficient. Finally, we must discuss the full details of the `step()` function. This function is responsible for estimating the model's objective function, updating the weights appropriately, and setting up the computation graph for the model. We start with the former.

The `step()` function consumes a number of arguments: the TensorFlow session, the list of vectors to feed as the encoder inputs, decoder inputs, target weights, the `bucket_id` selected during training, and the `forward_only` boolean flag, which will dictate whether or not we use gradient-based optimization to update the weights or to freeze them. Note that swapping this last flag from `False` to `True` is what allowed us to decode an arbitrary sentence and evaluate performance on a held-out set:

```
def step(self, session, encoder_inputs, decoder_inputs,
         target_weights, bucket_id, forward_only):
```

After some defensive checks to ensure that the vectors all have compatible sizes, we populate our input and output feeds. The input feed contains all the information initially passed to the `step()` function, which is all information needed to compute the overall loss per example:

```
 # Check if the sizes match.
encoder_size, decoder_size = self.buckets[bucket_id]
if len(encoder_inputs) != encoder_size:
  raise ValueError("Encoder length must be equal to the one
                    in bucket,"
                   " %d != %d." % (len(
                   encoder_inputs), encoder_size))
if len(decoder_inputs) != decoder_size:
  raise ValueError("Decoder length must be equal to the one
                    in bucket,"
                   " %d != %d." % (len(decoder_inputs),
                                   decoder_size))
if len(target_weights) != decoder_size:
  raise ValueError("Weights length must be equal to the one
                    in bucket,"
                   " %d != %d." % (len(target_weights),
                                   decoder_size))

# Input feed: encoder inputs, decoder inputs, target_weights,
# as provided.
input_feed = {}
for l in xrange(encoder_size):
  input_feed[self.encoder_inputs[l].name] = encoder_inputs[l]
for l in xrange(decoder_size):
  input_feed[self.decoder_inputs[l].name] = decoder_inputs[l]
  input_feed[self.target_weights[l].name] = target_weights[l]

# Since our targets are decoder inputs shifted by one,
# we need one more.
last_target = self.decoder_inputs[decoder_size].name
input_feed[last_target] = np.zeros([self.batch_size],
                                   dtype=np.int32)
```

The output feed, if a loss is computed and needs to be backpropagated through the network, contains the update operation that performs the stochastic gradient descent and computes the gradient norm and loss for the batch:

```
 # Output feed: depends on whether we do a backward step or
 # not.
if not forward_only:
  output_feed = [self.updates[bucket_id],  # Update Op that
                                           # does SGD.
                 self.gradient_norms[bucket_id],  # Gradient
                                                  # norm.
                 self.losses[bucket_id]]  # Loss for this
                                          # batch.
else:
  output_feed = [self.losses[bucket_id]]  # Loss for this
                                          # batch.
  for l in xrange(decoder_size):  # Output logits.
    output_feed.append(self.outputs[bucket_id][l])
```

These two feeds are passed to `session.run()`. Depending on the `forward_only` flag, either the gradient norm and loss are returned for maintaining statistics, or the outputs are returned for decoding purposes:

```
outputs = session.run(output_feed, input_feed)
if not forward_only:
  return outputs[1], outputs[2], None #, attns
    # Gradient norm, loss, no outputs.
else:
  return None, outputs[0], outputs[1:] #, attns
    # No gradient norm, loss, outputs.
```

Now, we can study the model itself. The constructor for the model sets up the computation graph using high-level constructs created. We first review the `create_model()` method briefly, which calls this constructor, and then discuss the details of this constructor.

The `create_model()` method itself is fairly straightforward: it uses a number of user-defined or default flags, such as the sizes of the English and French vocabularies and batch size, to create the model by using the constructor `seq2seq_model.Seq2SeqModel`. One particularly interesting flag is the `use_fp16` flag. With this, a lower precision is used as the type in the underlying numpy arrays; this results in faster performance at the cost of some amount of precision. However, it's often the case that 16-bit representations are sufficient for representing losses and gradient updates and often perform close to the level of using 32-bit representations. Model creation can be achieved using the following code:

```
def create_model(session, forward_only):
 """Create translation model and initialize or
    load parameters in session."""
 dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
 model = seq2seq_model.Seq2SeqModel(
     FLAGS.en_vocab_size,
     FLAGS.fr_vocab_size,
     _buckets,
     FLAGS.size,
     FLAGS.num_layers,
     FLAGS.max_gradient_norm,
     FLAGS.batch_size,
     FLAGS.learning_rate,
     FLAGS.learning_rate_decay_factor,
     forward_only=forward_only,
     dtype=dtype)
```

Before returning the model, a check is done to see if there are any previously checkpointed models from earlier training runs. If so, this model and its parameters are read into the model variable and used. This allows us to stop training at a checkpoint and later resume it without training from scratch. Otherwise, the fresh model created is returned as the main object:

```
    ckpt = tf.train.get_checkpoint_state(FLAGS.train_dir)
  if ckpt and tf.train.checkpoint_exists(
     ckpt.model_checkpoint_path):
    print("Reading model parameters from %s"
          % ckpt.model_checkpoint_path)
    model.saver.restore(session, ckpt.model_checkpoint_path)
  else:
    print("Created model with fresh parameters.")
    session.run(tf.global_variables_initializer())
  return model
```

We now review the constructor `seq2seq_model.Seq2SeqModel`. This constructor creates the entire computation graph and will occasionally call certain lower-level constructs. Before we jump to those details, we continue in our top-down investigation of the code and sketch the details of the overarching computation graph.

The same arguments passed to `create_model()` are passed to this constructor, and a few class-level fields are created:

```
class Seq2SeqModel(object):
  def __init__(self,
               source_vocab_size,
               target_vocab_size,
               buckets,
               size,
               num_layers,
               max_gradient_norm,
               batch_size,
               learning_rate,
               learning_rate_decay_factor,
               use_lstm=False,
               num_samples=512,
               forward_only=False,
               dtype=tf.float32):
    self.source_vocab_size = source_vocab_size
    self.target_vocab_size = target_vocab_size
    self.buckets = buckets
    self.batch_size = batch_size
    self.learning_rate = tf.Variable(
        float(learning_rate), trainable=False, dtype=dtype)
    self.learning_rate_decay_op = self.learning_rate.assign(
        self.learning_rate * learning_rate_decay_factor)
    self.global_step = tf.Variable(0, trainable=False)
```

The next part creates the sampled softmax and the output projection. This is an improvement over basic seq2seq models in that they allow for efficient decoding over large output vocabularies and project the output logits to the correct space:

```
# If we use sampled softmax, we need an output projection.
output_projection = None
softmax_loss_function = None
# Sampled softmax only makes sense if we sample less than
# vocabulary size.
if num_samples > 0 and num_samples <
  self.target_vocab_size:
  w_t = tf.get_variable("proj_w", [self.target_vocab_size,
                                   size], dtype=dtype)
  w = tf.transpose(w_t)
  b = tf.get_variable("proj_b", [self.target_vocab_size],
                      dtype=dtype)
  output_projection = (w, b)

  def sampled_loss(inputs, labels):
    labels = tf.reshape(labels, [-1, 1])
    # We need to compute the sampled_softmax_loss using
    # 32bit floats to avoid numerical instabilities.
    local_w_t = tf.cast(w_t, tf.float32)
    local_b = tf.cast(b, tf.float32)
    local_inputs = tf.cast(inputs, tf.float32)
    return tf.cast(
        tf.nn.sampled_softmax_loss(local_w_t, local_b,
                                   local_inputs, labels,
                                   num_samples,
                                   self.target_vocab_size),
        dtype)
  softmax_loss_function = sampled_loss
```

Based on the flags, we choose the underlying RNN cell, whether it's a GRU cell, an LSTM cell, or a multilayer LSTM cell. Production systems will rarely use single-layer LSTM cells, but they are much faster to train and may make the debugging cycle faster:

```
# Create the internal multi-layer cell for our RNN.
single_cell = tf.nn.rnn_cell.GRUCell(size)
if use_lstm:
  single_cell = tf.nn.rnn_cell.BasicLSTMCell(size)
cell = single_cell
if num_layers > 1:
  cell = tf.nn.rnn_cell.MultiRNNCell([single_cell] *
    num_layers)
```

The recurrent function `seq2seq_f()` is defined with `seq2seq.embedding_attention_seq2seq()`, which we will discuss later:

```
# The seq2seq function: we use embedding for the
# input and attention.
def seq2seq_f(encoder_inputs, decoder_inputs, do_decode):
  return seq2seq.embedding_attention_seq2seq(
      encoder_inputs,
      decoder_inputs,
```

```
        cell,
        num_encoder_symbols=source_vocab_size,
        num_decoder_symbols=target_vocab_size,
        embedding_size=size,
        output_projection=output_projection,
        feed_previous=do_decode,
        dtype=dtype)
```

We define placeholders for the inputs and targets:

```
    # Feeds for inputs.
    self.encoder_inputs = []
    self.decoder_inputs = []
    self.target_weights = []
    for i in xrange(buckets[-1][0]):  # Last bucket is
                                      # the biggest one.
      self.encoder_inputs.append(tf.placeholder(tf.int32,
                            shape=[None],
                            name="encoder{0}".format(i)))
    for i in xrange(buckets[-1][1] + 1):
      self.decoder_inputs.append(tf.placeholder(tf.int32,
                            shape=[None],
                            name="decoder{0}".format(i)))
      self.target_weights.append(tf.placeholder(dtype,
                            shape=[None],
                            name="weight{0}".format(i)))

    # Our targets are decoder inputs shifted by one.
    targets = [self.decoder_inputs[i + 1]
               for i in xrange(len(self.decoder_inputs) - 1)]
```

We now compute the outputs and losses from the function `seq2seq.model_with_buckets`. This function simply constructs the seq2seq model to be compatible with buckets and computes the loss either by averaging over the entire example sequence or as a weighted cross-entropy loss for a sequence of logits:

```
    # Training outputs and losses.
    if forward_only:
      self.outputs, self.losses = seq2seq.model_with_buckets(
          self.encoder_inputs, self.decoder_inputs, targets,
          self.target_weights, buckets, lambda x, y:
            seq2seq_f(x, y, True),
          softmax_loss_function=softmax_loss_function)
      # If we use output projection, we need to project outputs
      # for decoding.
      if output_projection is not None:
        for b in xrange(len(buckets)):
          self.outputs[b] = [
              tf.matmul(output, output_projection[0]) +
                output_projection[1]
              for output in self.outputs[b]
          ]
    else:
```

```
        self.outputs, self.losses = seq2seq.model_with_buckets(
            self.encoder_inputs, self.decoder_inputs, targets,
            self.target_weights, buckets,
            lambda x, y: seq2seq_f(x, y, False),
            softmax_loss_function=softmax_loss_function)
```

Finally, we update the parameters of the model (because they are trainable variables) using some form of gradient descent. We use vanilla SGD with gradient clipping, but we are free to use any optimizer—the results will certainly improve and training may proceed much faster. Afterward, we save all variables:

```
    # Gradients and SGD update operation for training the model.
    params = tf.trainable_variables()
    if not forward_only:
      self.gradient_norms = []
      self.updates = []
      opt = tf.train.GradientDescentOptimizer(
              self.learning_rate)
      for b in xrange(len(buckets)):
        gradients = tf.gradients(self.losses[b], params)
        clipped_gradients, norm = tf.clip_by_global_norm(
                                    gradients,
                                    max_gradient_norm)
        self.gradient_norms.append(norm)
        self.updates.append(opt.apply_gradients(
            zip(clipped_gradients, params), global_step=
                                            self.global_step))

    self.saver = tf.train.Saver(tf.all_variables())
```

With the high-level detail of the computation graph described, we now describe the last and lowest level of the model: the internals of `seq2seq.embedding_attention_seq2seq()`.

When initializing this model, several flags and arguments are passed as function arguments. One argument of particular note is `feed_previous`. When this is true, the decoder will use the outputted logit at time step T as input to time step T+1. In this way, it is sequentially decoding the next token based on all tokens thus far. We can describe this type of decoding, where the next output depends on all previous outputs, as *autoregressive decoding*:

```
def embedding_attention_seq2seq(encoder_inputs,
                                decoder_inputs,
                                cell,
                                num_encoder_symbols,
                                num_decoder_symbols,
                                embedding_size,
                                output_projection=None,
                                feed_previous=False,
                                dtype=None,
```

```
                                    scope=None,
                                    initial_state_attention=False):
```

We first create the wrapper for the encoder.

```
  with variable_scope.variable_scope(
      scope or "embedding_attention_seq2seq", dtype=dtype)
        as scope:
    dtype = scope.dtype
    encoder_cell = rnn_cell.EmbeddingWrapper(
        cell,
        embedding_classes=num_encoder_symbols,
        embedding_size=embedding_size)
    encoder_outputs, encoder_state = rnn.rnn(
        encoder_cell, encoder_inputs, dtype=dtype)
```

In this following code snippet, we calculate a concatenation of encoder outputs to put attention on; this is important because it allows the decoder to attend over these states as a distribution:

```
    # First calculate a concatenation of encoder outputs
    # to put attention on.
    top_states = [
        array_ops.reshape(e, [-1, 1, cell.output_size]) for e
         in encoder_outputs
    ]
    attention_states = array_ops.concat(1, top_states)
```

Now, we create the decoder. If the `output_projection` flag is not specified, the cell is wrapped to be one that uses an output projection:

```
    output_size = None
    if output_projection is None:
      cell = rnn_cell.OutputProjectionWrapper(cell,
               num_decoder_symbols)
      output_size = num_decoder_symbols
```

From here, we compute the outputs and states using the `embedding_attention_decoder`:

```
    if isinstance(feed_previous, bool):
      return embedding_attention_decoder(
          decoder_inputs,
          encoder_state,
          attention_states,
          cell,
          num_decoder_symbols,
          embedding_size,
          output_size=output_size,
          output_projection=output_projection,
          feed_previous=feed_previous,
          initial_state_attention=initial_state_attention)
```

The `embedding_attention_decoder` is a simple improvement over the `attention_decoder` described in the previous section; essentially, the inputs are projected to a learned embedding space, which usually improves performance. The `loop` function, which simply describes the dynamics of the recurrent cell with embedding, is invoked in this step:

```
def embedding_attention_decoder(decoder_inputs,
                                initial_state,
                                attention_states,
                                cell,
                                num_symbols,
                                embedding_size,
                                output_size=None,
                                output_projection=None,
                                feed_previous=False,
                                update_embedding_for_previous=
                                 True,
                                dtype=None,
                                scope=None,
                                initial_state_attention=False):

  if output_size is None:
    output_size = cell.output_size
  if output_projection is not None:
    proj_biases = ops.convert_to_tensor(output_projection[1],
      dtype=dtype)
    proj_biases.get_shape().assert_is_compatible_with(
        [num_symbols])

  with variable_scope.variable_scope(
      scope or "embedding_attention_decoder", dtype=dtype)
        as scope:

    embedding = variable_scope.get_variable("embedding",
                                            [num_symbols,
                                             embedding_size])
    loop_function = _extract_argmax_and_embed(
        embedding, output_projection,
        update_embedding_for_previous) if feed_previous
          else None
    emb_inp = [
        embedding_ops.embedding_lookup(embedding, i) for i in
          decoder_inputs
    ]
    return attention_decoder(
        emb_inp,
        initial_state,
        attention_states,
        cell,
        output_size=output_size,
```

```
        loop_function=loop_function,
        initial_state_attention=initial_state_attention)
```

The last step is to study the `attention_decoder` itself. As the name suggests, the main feature of this decoder is that it computes a set of attention weights over the hidden states that the encoder emitted during encoding. After defensive checks, we reshape the hidden features to the right size:

```
def attention_decoder(decoder_inputs,
                      initial_state,
                      attention_states,
                      cell,
                      output_size=None,
                      loop_function=None,
                      dtype=None,
                      scope=None,
                      initial_state_attention=False):
  if not decoder_inputs:
    raise ValueError("Must provide at least 1 input to attention
                      decoder.")
  if attention_states.get_shape()[2].value is None:
    raise ValueError("Shape[2] of attention_states must be known:
      %s" %
                     attention_states.get_shape())
  if output_size is None:
    output_size = cell.output_size

  with variable_scope.variable_scope(
      scope or "attention_decoder", dtype=dtype) as scope:
    dtype = scope.dtype

    batch_size = array_ops.shape(decoder_inputs[0])[0]  # Needed
                                                        # for
                                                      #reshaping.
    attn_length = attention_states.get_shape()[1].value
    if attn_length is None:
      attn_length = array_ops.shape(attention_states)[1]
    attn_size = attention_states.get_shape()[2].value

    # To calculate W1 * h_t we use a 1-by-1 convolution,
    # need to reshape before.
    hidden = array_ops.reshape(attention_states,
                               [-1, attn_length, 1, attn_size])
    hidden_features = []
    v = []
    attention_vec_size = attn_size  # Size of query vectors
      for attention.
    k = variable_scope.get_variable("AttnW_0",
                                    [1, 1, attn_size,
                                     attention_vec_size])
    hidden_features.append(nn_ops.conv2d(hidden, k,
      [1, 1, 1, 1], "SAME"))
```

```
    v.append(
        variable_scope.get_variable("AttnV_0",
                                    [attention_vec_size]))

    state = initial_state
```

We now define the `attention()` method itself, which consumes a query vector and returns the attention-weighted vector over the hidden states. This method implements the same attention as described in the previous section:

```
def attention(query):
  """Put attention masks on hidden using hidden_features
   and query."""
  ds = []  # Results of attention reads will be
           # stored here.
  if nest.is_sequence(query):  # If the query is a tuple,
                               # flatten it.
    query_list = nest.flatten(query)
    for q in query_list:  # Check that ndims == 2 if
                          # specified.
      ndims = q.get_shape().ndims
      if ndims:
        assert ndims == 2
    query = array_ops.concat(1, query_list)
    # query = array_ops.concat(query_list, 1)
  with variable_scope.variable_scope("Attention_0"):
    y = linear(query, attention_vec_size, True)
    y = array_ops.reshape(y, [-1, 1, 1,
         attention_vec_size])
    # Attention mask is a softmax of v^T * tanh(...).
    s = math_ops.reduce_sum(v[0] * math_ops.tanh(
                            hidden_features[0] + y),
                            [2, 3])
    a = nn_ops.softmax(s)
    # Now calculate the attention-weighted vector d.
    d = math_ops.reduce_sum(
        array_ops.reshape(a, [-1, attn_length, 1, 1]) *
                                    hidden, [1, 2])
    ds.append(array_ops.reshape(d, [-1, attn_size]))
  return ds
```

Using the function, we compute the attention over each of the output states, starting with the initial state:

```
outputs = []
prev = None
batch_attn_size = array_ops.stack([batch_size, attn_size])
attns = [array_ops.zeros(batch_attn_size, dtype=dtype)]
for a in attns:  # Ensure the second shape of attention
                 # vectors is set.
  a.set_shape([None, attn_size])
if initial_state_attention:
  attns = attention(initial_state)
```

Now we loop over the rest of the inputs. We perform a defensive check to ensure that the input at the current time step is the right size. Then we run the RNN cell as well as the attention query. These two are then combined and passed to the output according to the same dynamics:

```
    for i, inp in enumerate(decoder_inputs):
      if i > 0:
        variable_scope.get_variable_scope().reuse_variables()
      # If loop_function is set, we use it instead of
      # decoder_inputs.
      if loop_function is not None and prev is not None:
        with variable_scope.variable_scope("loop_function",
          reuse=True):
          inp = loop_function(prev, i)
      # Merge input and previous attentions into one vector of
      # the right size.
      input_size = inp.get_shape().with_rank(2)[1]
      if input_size.value is None:
        raise ValueError("Could not infer input size from input:
          %s" % inp.name)
      x = linear([inp] + attns, input_size, True)
      # Run the RNN.
      cell_output, state = cell(x, state)
      # Run the attention mechanism.
      if i == 0 and initial_state_attention:
        with variable_scope.variable_scope(
            variable_scope.get_variable_scope(), reuse=True):
          attns = attention(state)
      else:
        attns = attention(state)

      with variable_scope.variable_scope(
        "AttnOutputProjection"):
        output = linear([cell_output] + attns, output_size,
                  True)
      if loop_function is not None:
        prev = output
      outputs.append(output)

  return outputs, state
```

With this, we've successfully completed a full tour of the implementation details of a fairly sophisticated neural machine translation system. Production systems have additional tricks that are not as generalizable, and these systems are trained on huge compute servers to ensure that state-of-the-art performance is met.

For reference, this exact model was trained on eight NVIDIA Telsa M40 GPUs for four days. We show plots for the perplexity in Figure 7-31 and Figure 7-32, and show the learning rate anneal over time as well.

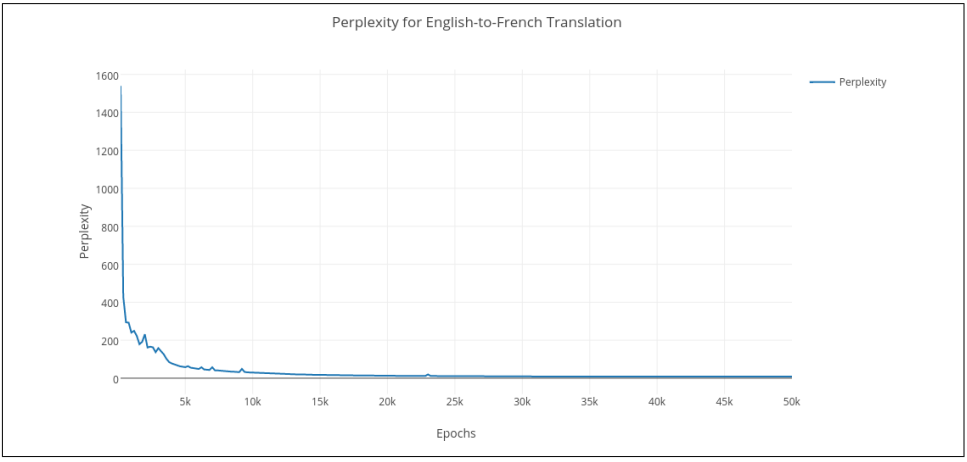


Figure 7-31. Plot of perplexity on training data over time. After 50k epochs, the perplexity decreases from about 6 to 4, which is a reasonable score for a neural machine translation system.

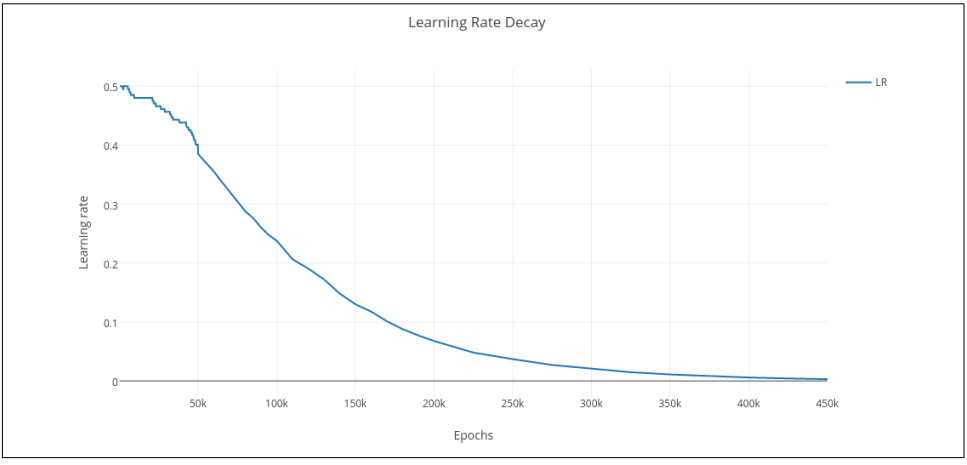


Figure 7-32. Plot of learning rate over time; as opposed to perplexity, we observe that the learning rate almost smoothly declines to 0. This means that by the time we stopped training, the model was approaching a stable state.

To showcase the attentional model more explicitly, we can visualize the attention that the decoder LSTM computes while translating a sentence from English to French. In particular, we know that as the encoder LSTM is updating its cell state in order to compress the sentence into a continuous vector representations, it also computes hidden states at every time step. We know that the decoder LSTM computes a convex sum over these hidden states, and one can think of this sum as the attention mecha-

nism; when there is more weight on a particular hidden state, we can interpret that as the model is paying more attention to the token inputted at that time step.

This is exactly what we visualize in Figure 7-33. The English sentence to be translated is on the top row, and the resulting French translation is on the first column. The lighter a square is, the more attention the decoder paid to that particular column when decoding that row element. That is, the (i, j)th element in the attention map shows the amount of attention that was paid to the jth token in the English sentence when translating the ith token in the French sentence.

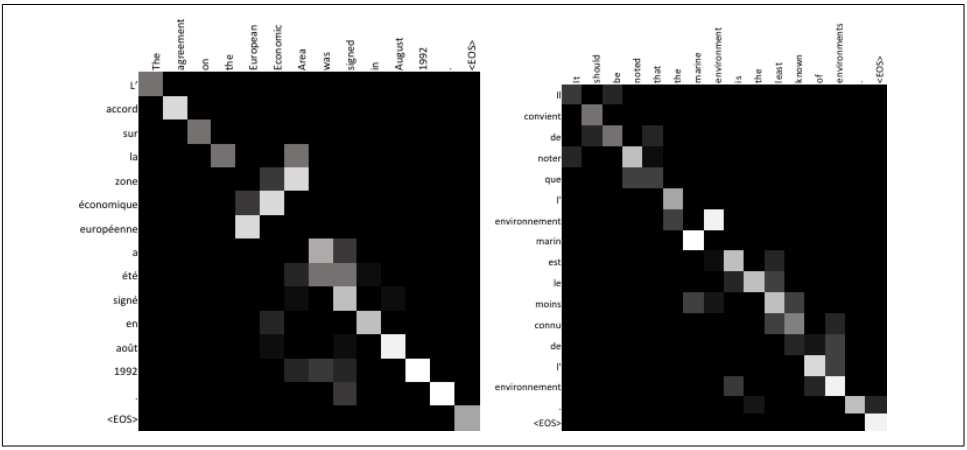


Figure 7-33. We can explicitly visualize the weights of the convex sum when the decoder attends over hidden states in the encoder. The lighter the square, the more attention was placed on that element.

We can immediately see that the attention mechanism seems to be working quite well. Large amounts of attention are generally being placed in the right areas, even though there is slight noise in the model's prediction. It is possible that adding additional layers to the network would help produce crisper attention. One impressive aspect is that the phrase "the European Economic" is translated in reverse in French as the "zone économique européenne," and as such, the attention weights reflect this flip! These kinds of attention patterns may be even more interesting when translating from English to a different language that does not parse smoothly from left to right.

With one of the most fundamental architectures understood and implemented, we now move forward to study exciting new developments with recurrent neural networks and begin a foray into more sophisticated learning.

Summary

In this chapter, we've delved deep into the world of sequence analysis. We've analyzed how we might hack feed-forward networks to process sequences, developed a strong understanding of recurrent neural networks, and explored how attentional mechanisms can enable incredible applications ranging from language translation to audio transcription.

CHAPTER 8

Memory Augmented Neural Networks

Mostafa Samir[1] *and Surya Bhupatiraju*

So far we've seen how effective an RNN can be at solving a complex problem like machine translation. However, we're still far from reaching its full potential! In Chapter 7 we mentioned that it's theoretically proven that the RNN architecture is a universal functional representer; a more precise statement of the same result is that RNNs are *Turing complete*. This simply means that given proper wiring and adequate parameters, an RNN can learn to solve any computable problem, which is basically any problem that can be solved by a computer algorithm or, equivalently, a Turing machine.

Neural Turing Machines

Though theoretically possible, it's extremely difficult to achieve that kind of universality in practice! This difficulty stems from the fact that we're looking at an immensely huge search space of possible wirings and parameter values of RNNs, a space so vastly large for gradient descent to find an appropriate solution for any arbitrary problem. However, in the remaining sections of this chapter we'll start exploring some approaches at the edge of research that would allow us to start tapping into that potential!

Let's think for a while about a very simple reading comprehension question like the following:

```
Mary travelled to the hallway. She grabbed the milk glass there.
Then she travelled to the office, where she found an apple
and grabbed it.
```

1 *https://mostafa-samir.github.io/*

How many objects is Mary carrying?

The answer is so trivial: it's two! But what actually happened in our brains that allowed us to come up with the answer so trivially? If we thought about how we could solve that comprehension question using a simple computer program, our approach would probably go like this:

```
1. allocate a memory location for a counter
2. initialize counter to 0
3. for each word in passage
    3.1. if word is 'grabbed'
        3.1.1. increment counter
4. return counter value
```

It turns out that our brains tackle the same task in a very similar way to that simple computer program. Once we start reading, we start allocating memory (just as our computer program) and store the pieces of information we receive. We start by storing that location of Mary, which after the first sentence is the hallway. In the second sentence we store the objects Mary is carrying, and by now it's only a glass of milk. Once we see the third sentence, our brain modifies the first memory location to point to the office. By the end of the fourth sentence, the second memory location is modified to include both the milk and the apple. When we finally encounter the question, our brains quickly query the second memory location and count the information there, which turns out to be two! In neuroscience and cognitive psychology, such a system of transient storing and manipulation of information is called a *working memory*, and it's the main inspiration behind the line of research we'll be discussing in the rest of this chapter.

In 2014, Graves et al. from Google DeepMind started this line of work in a paper called "Neural Turing Machines" in which they introduced a new neural architecture with the same name, a *Neural Turing Machine* (or NTM), that consists of a controller neural network (usually an RNN) with an external memory that resembles the brain's working memory. For the close resemblance between the working memory model and the computer model we just saw, Figure 8-1 shows that the same resemblance holds for the NTM architecture, with the external memory in place of the RAM, the read/write heads in place of the read/write buses, and the controller network in place of the CPU, except for the fact that the controller learns its program, unlike the CPU, which is fed its program.

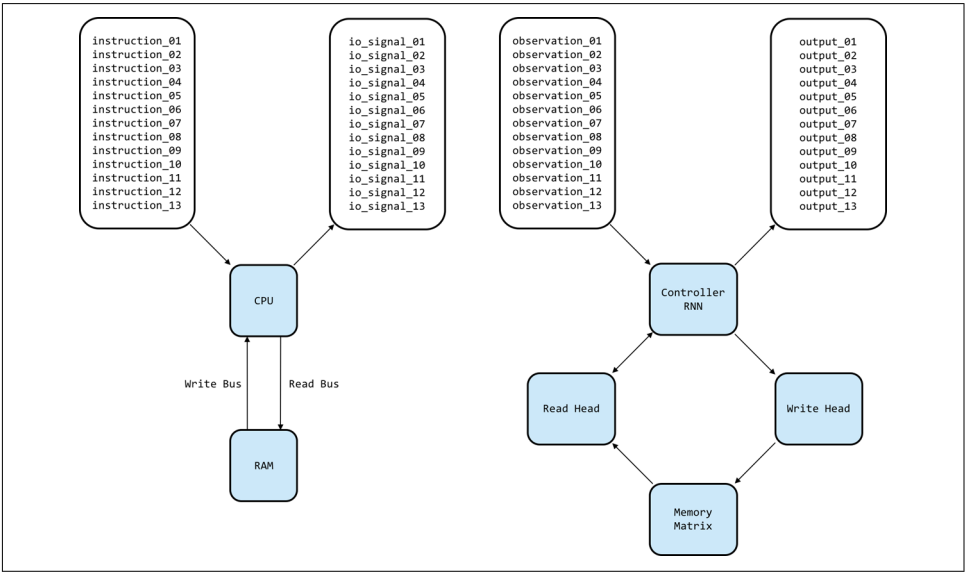


Figure 8-1. Comparing the architecture of a modern day computer which is fed its program (left) to a Neural Turing Machine that learns its program (right). This example has a single read head and single write head, but an NTM can have several in practice.

If we thought about NTMs in light of our earlier discussion of RNN's Turing completeness, we'll find that augmenting the RNN with an external memory for transient storage prunes a large portion out of that search space, as we now don't care about exploring RNNs that can both process and store the information; we're just looking for the RNNs that can process the information stored outside of it. This pruning of the search space allows us to start tapping into some of the RNN potentials that were locked away before augmenting it with a memory, evident by the variety of tasks that the NTM could learn: from copying input sequences after seeing them, to emulating N-gram models, to performing a priority sort on data. We'll even see by the end of the chapter how an extension to the NTM can learn to do reading comprehension tasks like the one we saw earlier, with nothing more than a gradient-based search!

Attention-Based Memory Access

To be able to train an NTM with a gradient-based search method, we need to make sure that the whole architecture is differentiable so that we can compute the gradient of some output loss with respect to the model's parameters that process the input. This property is called *end-to-end-differentiable*, with one end being the inputs and the other the outputs. If we attempted to access the NTM's memory in the same way a digital computer accesses its RAM, via discrete values of addresses, the discreteness of the addresses would introduce discontinuities in gradients of the output, and hence

we would lose ability to train the model with a gradient-based method. We need a continuous way to access the memory while being able to "focus" on a specific location in it. This kind of continuous focusing can be achieved via attention methods!

Instead of generating a discrete memory address, we let each head generate a normalized softmax attention vector with the same size as the number of memory locations. With this attention vector, we'll be accessing all the memory locations at the same time in a blurry manner, with each value in the vector telling us how much we're going to focus on the corresponding location, or how likely we're going to access it. For example, to read a vector at a time step t out of our $N \times W$ NTM's memory matrix denoted by M_t (where N is the number of locations and W is the size of the location), we generate an attention vector, or a weighting vector w_t of size N, and our read vector can be calculated via the product:

$$\mathbf{r}_t = M_t^\top w_t$$

where $^\top$ denotes the matrix transpose operation. Figure 8-2 shows how with the weights attending to a specific location, we can retrieve a read vector that approximately contains the same information as the content of that memory location.

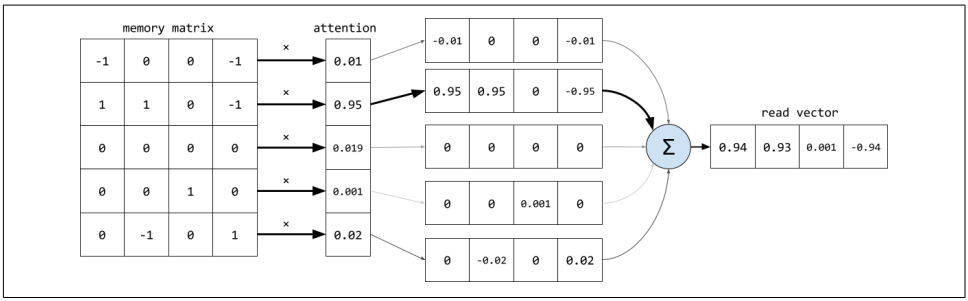


Figure 8-2. A demonstration of how a blurry attention-based reading can retrieve a vector containing approximately the same information as in the focused-on location

A similar attention weighting method is used for the write head: a weighting vector w_t is generated and used for erasing specific information from the memory, as specified by the controller in an erase vector e_t that has W values between 0 and 1 specifying what to erase and to what keep. Then we use the same weighting for writing to the erased memory matrix some new information, also specified by the controller in a write vector v_t containing W values:

$$M_t = M_{t-1} \circ \left(E - w_t e_t^\top\right) + w_t v_t^\top$$

where E is a matrix of ones and $\circ$ is element-wise multiplication. Similar to the reading case, the weighting w_t tells us where to focus our erasing (the first term of the equation) and writing operations (the second term).

NTM Memory Addressing Mechanisms

Now that we understand how NTMs access their memories in a continuous manner via attention weighting, we're left with how these weightings are generated and what forms of memory addressing mechanisms they represent. We can understand that by exploring what NTMs are expected to do with their memories, and based on the model they are mimicking (the Turning machine), we expect them to be able access a location by the value it contains, and to be able to go forward or backward from a given location.

The first mode of behavior can be achieved with an access mechanism that we'll call *content-based addressing*. In this form of addressing, the controller emits the value that it's looking for, which we'll call a key k_t, then it measures its similarity to the information stored in each location and focuses the attention on the most similar one. This kind of weighting can be calculated via:

$$\mathscr{C}(M,k,\beta) = \frac{\exp\left(\beta \mathscr{D}(M,k)\right)}{\sum_{i=0}^{N} \exp\left(\beta \mathscr{D}(M[i],k)\right)}$$

where $\mathscr{D}$ is some similarity measure, like the cosine similarity. The equation is nothing more than a normalized softmax distribution over the similarity scores. There is, however, an extra parameter β that is used to attenuate the attention weights if needed. We call that the key strength. The main idea behind that parameter is that for some tasks, the key emitted by the controller may not be very close to any of the information in the memory which would result in seemingly uniform attention weights. Figure 8-3 shows how the key strength allows the controller to learn how to attenuate such uniform attention to be more focused on a single location that is the most probable; the controller then learns what value of the strength to emit with each possible key it emits.

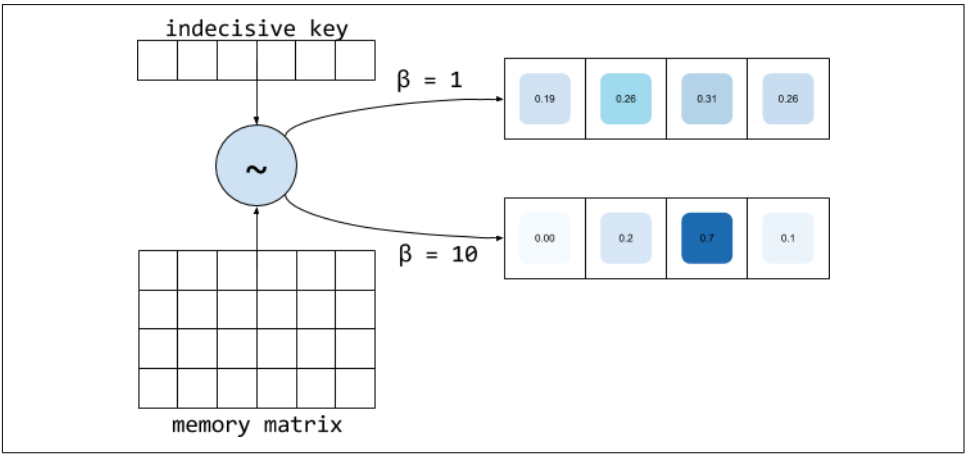


Figure 8-3. An indecisive key with unit strength results in a nearly-uniform attention vector, which isn't helpful. Increasing the strength for keys like that focuses the attention on the most probable location.

To move forward and backward in the memory, we first need to know where are we standing now, and such information is located in the access weighting from the last time step w_{t-1}. So to preserve the information about our current location with the new content-based weighting w_t^c we just got, we interpolate between the two weighting using a scalar g_t that lies between 0 and 1:

$$w_t^g = g_t w_t^c + (1 - g_t) w_{t-1}$$

We call g_t the *interpolation gate*, and it's also emitted by the controller to control the kind of information we want to use in the current time step. When the gate's value is close to 1, we favor the addressing given by content lookup. However, when it's close to 0, we tend to pass the information about our current location through and ignore the content-based addressing. The controller learns to use this gate so that, for example, it could set it 0 when iteration through consecutive locations is desired and information about the current location is crucial. The type of information the controller chooses to gate through is denoted by the *gated weighting* w_t^g.

To start moving around the memory we need a way to take our current gated weighting and shift the focus from one location to another. This can be done via convoluting the gated weighting with a *shift weighting* s_t also emitted by the controller. This shift weighting is a normalized softmax attention vector of size $n + 1$, where n is an even integer specifying the number of possible shifts around the focused-on location in the gated weighting; for example, if it has a size of 3, then there are two possible shifts around a location: one forward and one backward. Figure 8-4 shows how a shift weighting can move around the focused-on location in gated weighting. The shifting

occurs via convoluting the gated weighting by the shift weighting in pretty much the same way we convoluted images with feature maps back in Chapter 5. The only exception is in how we handle the case when the shift weightings go outside the gated weighting. Instead of using padding like we did before, we use a rotational convolution operator where overflown weights get applied to the values at the other end of the gated weighting, as shown in middle panel of Figure 8-4. This operation can be expressed element-wise as:

$$\tilde{w}_t[i] = \sum_{j=0}^{|s_t|} w_t^g \left[\left(i + \frac{|s_t| - 1}{2} - j \right) \mod N \right] s_t[j]$$

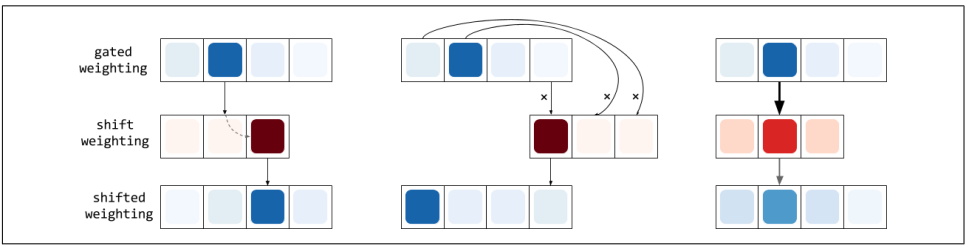


Figure 8-4. (left) A shift weighting focused on the right shifts the gated weighting one location to the right. (middle) Rotational convolution on a left-focused shift weighting, shifting the gated weighting to the left. (right) A nonsharp centered shift weighting keeps the gated weighting intact but disperses it.

With the introduction of the shifting operation, our heads' weightings can now move around the memory freely forward and backward. However, a problem occurs if at any time the shift weighting is not sharp enough. Because of the nature of the convolution operation, a nonsharp shift weighting (as in the right panel of Figure 8-4) disperses the original gated weightings around its surroundings and results in a less focused shifted weighting. To overcome that blurring effect, we run the shifted weightings through one last operation: a sharpening operation. The controller emits one last scalar $\gamma_t \geq 1$ that sharpens the shifted weightings via:

$$w_t = \frac{\tilde{w}_t^{\gamma_t}}{\sum_{i=0}^{N} \tilde{w}_t[i]^{\gamma_t}}$$

Starting from interpolation down to the final weighting vector out of sharpening, this process constitutes the second addressing mechanism of NTMs: the *location-based mechanism*. Using a combination of both addressing mechanisms, an NTM is able to utilize its memory to learn to solve various tasks. One of these tasks that would allow us to get a deeper look into the NTM in action is the copy task shown in Figure 8-5. In this task, we present the model with a sequence of random binary vectors that ter-

minate with a special end symbol. We then request the same input sequence to be copied to the output.

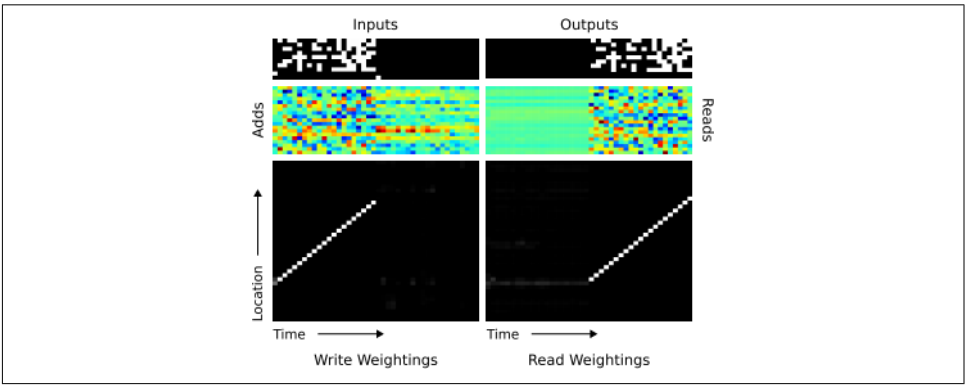


Figure 8-5. A visualization of an NTM trained on the copy task. (left) From top to bottom it shows the model's input, write vectors, and the write weightings across the memory locations through time. (right) From top to bottom it shows the model's output, read vectors, and read weighting across the memory locations through time. Source: Graves et al. "Neural turing machines." (2014)

The visualization shows how at the input time, the NTM starts writing the inputs step by step into consecutive locations in the memory. In the output time, the NTM goes back at the first written vector and iterates through the next locations to read and return the previously written input sequence. The original NTM paper contains several other visualizations of NTMs trained on different problems which are worth checking. These visualizations demonstrate the architecture's ability to utilize the addressing mechanisms to adapt to and learn to solve various tasks.

We'll suffice with our current understanding of NTMs and skip its implementation. Instead, we will spend the rest of the chapter exploring the drawbacks of NTMs and how the novel architecture of Differentiable Neural Computer (DNC) was able to overcome these drawbacks. We'll conclude our discussion by implementing that novel architecture on simple reading comprehension tasks like the one we saw earlier.

Differentiable Neural Computers

Despite the power of NTMs, they have a few limitations regarding their memory mechanisms. The first of these limitations is that NTMs have no way to ensure that no interference or overlap between written data would occur. This is due to the nature of the "differentiable" writing operation in which we write new data everywhere in the memory to some extent specified by the attention. Usually, the attention mechanisms learn to focus the write weightings strongly on a single memory loca-

tion, and the NTM converges to a mostly interference-free behavior, but that's not guaranteed.

However, even when the NTM converges to an interference-free behavior, once a memory location has been written to, there's no way to reuse that location again, even when the data stored in it becomes irrelevant. The inability to free and reuse memory locations is the second limitation to the NTM architecture. This results in new data being written to new locations which are likely to be contiguous, as we saw with the copy task. This contiguous writing fashion is the only way for an NTM to record any temporal information about the data being written: consecutive data is stored in consecutive locations. If the write head jumped to another place in the memory while writing some consecutive data, a read head won't be able to recover the temporal link between the data written before and after the jump: this constitutes the third limitation of NTMs.

In October 2016, Graves et al. from DeepMind published in *Nature* a paper titled "Hybrid computing using a neural network with dynamic external memory" in which they introduced a new memory-augmented neural architecture called *differentiable neural computer* (DNC) that improves on NTMs and addresses those limitations we just discussed. Similar to NTMs, DNCs consists of a controller that interacts with an external memory. The memory consists of N words of size W, making up an $N \times W$ matrix we'll be calling M. The controller takes in an input vector of size X and the R vectors of size W read from memory in the previous step, where R is the number of read heads. The controller then processes them through a neural network, then returns two pieces of information:

- An *interface vector* that contains all the necessary information to query the memory (i.e., write and read from it)
- A *pre-output* vector of size Y

The external memory then takes in the interface vector, performs the necessary writing through a single write head, then reads R new vectors from the memory. It returns the newly read vectors to the controller to be added with the pre-output vector, producing the final output vector of size Y.

Figure 8-6 summarizes the operation of the DNC that we just described. We can see that unlike NTMs, DNCs keep other data structures alongside the memory itself to keep track of the state of the memory. As we'll shortly see, with these data structures and some clever new attention mechanisms, DNCs are able to successfully overcome NTM's limitations.

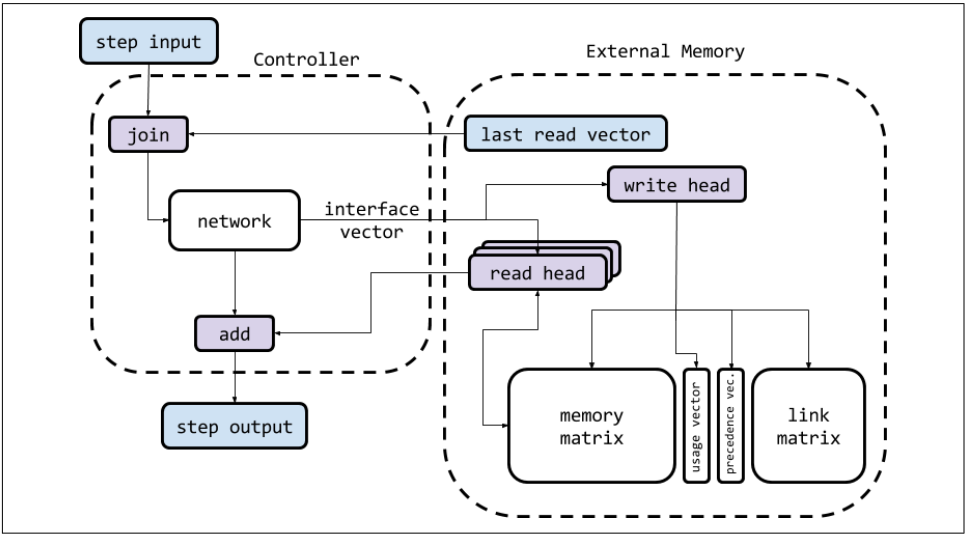


Figure 8-6. An overview of DNC's architecture and operation.
DNC's external memory differs from that of an NTM by several extra data structures as well as by the attention mechanisms used to access the memory.

To make the whole architecture differentiable, DNCs access the memory through weight vectors of size N whose elements determine how much the heads focus on each memory location. There are R weightings for the read heads $\mathrm{w}_t^{r,1}, \cdots, \mathrm{w}_t^{r,R}$ where t denotes the time step. On the other hand, there's one write weighting w_t^{w} for the single write head. Once we obtain these weightings, we can modify the memory matrix and get updated via:

$$M_t = M_{t-1} \circ \left(E - \mathrm{w}_t^{w} e_t^{\top}\right) + \mathrm{w}_t^{w} \mathrm{v}_t^{\top}$$

e_t, v_t are the *erase* and *write* vectors we saw earlier with NTMs, coming from the controller through the interface vector as instructions about what to erase from and write to the memory.

As soon as we get the updated memory matrix M_t, we can read out the new read vectors $\mathrm{r}_t^1, \mathrm{r}_t^2, \cdots, \mathrm{r}_t^R$ using the following equation for each read weighting:

$$\mathrm{r}_t^i = M_t^{\top} \mathrm{w}_t^{r,i}$$

Up until now, it seems that there's nothing different from how NTMs write to and read from memory. However, the differences will start to show up when we discuss the attention mechanisms DNCs use to obtain their access weightings. While they both share the content-based addressing mechanism $\mathscr{C}(M, k, \beta)$ defined earlier, DNCs use more sophisticated mechanisms to attend more efficiently to the memory.

Interference-Free Writing in DNCs

The first limitation we discussed of NTMs was their inability to ensure an interference-free writing behavior. An intuitive way to address this issue is to design the architecture to focus strongly on a single, free memory location and not wait for NTM to learn to do so. In order to keep track of which locations are free and which are busy, we need to introduce a new data structure that can hold this kind of information. We'll call it the *usage vector.*

The usage vector u_t is a vector of size N where each element holds a value between 0 and 1 that represents how much the corresponding memory location is used; with 0 indicating a completely free location and 1 indicating a completely used one.

The usage vector initially contains zeros $\mathrm{u}_0 = \mathbf{0}$ and gets updated with the usage information across the steps. Using this information, it's clear that the location to which the weights should attend most strongly to is the one with the least usage value. To obtain such weighting, we need first to sort the usage vector and obtain the list of location indices in ascending order of the usage; we call such a list a *free list* and denote it by ϕ_t. Using that free list, we can construct an intermediate weighting called the *allocation weighting* a_t that would determine which memory location should be allocated for new data. We calculate a_t using:

$$\mathrm{a}_t[\phi_t[j]] = (1 - \mathrm{u}_t[\phi_t[j]])\prod_{i=1}^{j-1} \mathrm{u}_t[\phi_t[i]] \quad \text{where } j \in 1, \cdots, N$$

This equation may look incomprehensible at first glance. A good way to understand it is to work through it with a numerical example, for example, when $\mathrm{u}_t = [1, 0.7, 0.2, 0.4]$. We'll leave the details for you to go through. In the end you should arrive at the allocation weighting being $\mathrm{a}_t = [0, 0.024, 0.8, 0.12]$. As we go through the calculations, we'll begin to understand how this formula works: the $1 - \mathrm{u}_t[\phi_t[j]]$makes the location weight proportional to how free it is. By noticing that the product $\prod_{i=1}^{j-1} \mathrm{u}_t[\phi_t[j]]$ gets smaller and smaller as we iterate through the free list (because we keep multiplying small values between 0 and 1), we can see that this product decreases the location weight even more as we go from the least used location to the most used one, which finally results in the least used location having the largest weight, while the most used one gets the smallest weight. So we're able to guarantee the ability to focus on a single location by design without the the need to hope for the model to learn it on its own from scratch; this means more reliability as well as faster training time.

With the allocation weighting a_t and lookup weighting c_t^w we get from the content-based addressing mechanism $\mathrm{c}_t^w = \mathscr{C}(M_{t-1}, k_t^w, \beta_t^w)$ where k_t^w, β_t^w are the lookup key and the lookup strength we receive through the interface vector, we can now construct our final write weighting:

$$\mathrm{w}_t^w = g_t^w[g_t^a \mathrm{a}_t + (1 - g_t^a)\mathrm{c}_t^w]$$

where g_t^w, g_t^a are values between 0 and 1 called the write and allocation gates, which we also get from the controller through the interface vector. These gates control the writing operation with g_t^w determining if any writing is going to happen in the first place, and g_t^a specifying whether we'll write to a new location using the allocation weighting or modify an existing value specified by the lookup weighting.

DNC Memory Reuse

What if while we calculate the allocation weighting we find that all locations are used, or in other words $\mathrm{u}_t = \mathbf{1}$? This means that the allocation weightings will turn out all zeros and no new data can be allocated to memory. This raises the need for the ability to free and reuse the memory.

In order to know which locations can be freed and which cannot, we construct a *retention vector* ψ_t of size N that specifies how much of each location should be retained and not get freed. Each element of this vector takes a value between 0 and 1, with 0 indicating that the corresponding location can be freed and 1 indicating that it should be retained. This vector is calculated using:

$$\psi_t = \Pi_{i=1}^{R}\left(\mathbf{1} - f_t^i \mathrm{w}_{t-1}^{r,i}\right)$$

This equation is basically saying that the degree to which a memory location should be freed is proportional to how much is read from it in the last time steps by the various read heads (represented by the values of the read weightings $\mathrm{w}_{t-1}^{r,i}$). However, continuously freeing a memory location once its data is read is not generally preferable as we might still need the data afterward. We let the controller decide when to free and when to retain a location after reading by emitting a set of R free gates $f_t^1, \cdots, f_t^R$ that have a value between 0 and 1. This determines how much freeing should be done based on the fact that the location was just read from. The controller will then learn how to use these gates to achieve the behavior it desires.

Once the retention vector is obtained, we can use it to update the usage vector to reflect any freeing or retention made via:

$$\mathrm{u}_t = \left(\mathrm{u}_{t-1} + \mathrm{w}_{t-1}^w - \mathrm{u}_{t-1} \circ \mathrm{w}_{t-1}^w\right) \circ \psi_t$$

This equation can be read as follows: a location will be used if it has been retained (its value in $\psi_t \approx 1$) and either it's already in use or has just been written to (indicated by its value in $\mathrm{u}_{t-1} + \mathrm{w}_{t-1}^w$). Subtracting the element-wise product $\mathrm{u}_{t-1} \circ \mathrm{w}_{t-1}^w$brings the whole expression back between 0 and 1 to be a valid usage value in case the addition between the previous usage got the write weighting past 1.

By doing this usage update step before calculating the allocation, we can introduce some free memory for possible new data. We're also able to use and reuse a limited amount of memory efficiently and overcome the second limitation of NTMs.

Temporal Linking of DNC Writes

With the dynamic memory management mechanisms that DNCs use, each time a memory location is requested for allocation, we're going to get the most unused location, and there'll be no positional relation between that location and the location of the previous write. With this type of memory access, NTM's way of preserving temporal relation with contiguity is not suitable. We'll need to keep an explicit record of the order of the written data.

This explicit recording is achieved in DNCs via two additional data structures alongside the memory matrix and the usage vector. The first is called a *precedence vector* p_t, an N-sized vector considered to be a probability distribution over the memory locations, with each value indicating how likely the corresponding location was the last one written to. The precedence is initially set to zero $\mathrm{p}_0 = \mathbf{0}$ and gets updated in the following steps via:

$$\mathrm{p}_t = \left(1 - \Sigma_{i=1}^{N} \mathrm{w}_t^w[i]\right)\mathrm{p}_{t-1} + \mathrm{w}_t^w$$

Updating is done by first resetting the previous values of the precedence with a reset factor that is proportionate to how much writing was just made to the memory (indicated by the summation of the write weighting's components). Then the value of write weighting is added to the reset value so that a location with a large write weighting (that is the most recent location written to) would also get a large value in the precedence vector.

The second data structure we need to record temporal information is the *link matrix* L_t. The link matrix is an $N \times N$ matrix in which the element $\mathrm{L}_t[i, j]$ has a value between 0,1, indicating how likely it is that location i was written after location j. This matrix is also initialized to zeros, and the diagonal elements are kept at zero throughout the time $\mathrm{L}_t[i, i] = 0$, as it's meaningless to track if a location was written after itself when the previous data has already been overwritten and lost. However, each other element in the matrix is updated using:

$$\mathrm{L}_t[i, j] = \left(1 - \mathrm{w}_t^w[i] - \mathrm{w}_t^w[j]\right)\mathrm{L}_{t-1}[i, j] + \mathrm{w}_t^w[i]\mathrm{p}_{t-1}[j]$$

The equation follows the same pattern we saw with other update rules: first the link element is reset by a factor proportional to how much writing had been done on locations i, j. Then the link is updated by the correlation (represented here by multiplication) between the write weighting at location i and the previous precedence value of

location j. This eliminates NTM's third limitation; now we can keep track of temporal information no matter how the write head hops around the memory.

Understanding the DNC Read Head

Once the write head has finished updating the memory matrix and the associated data structures, the read head is now ready to work. Its operation is simple: it needs to be able to look up values in the memory and be able to iterate forward and backward in temporal ordering between data. The lookup ability can simply be achieved with content-based addressing: for each read head, i we calculate an intermediate weighting $\mathrm{c}_t^{r,i} = \mathscr{C}\left(M_t, k_t^{r,i}, \beta_t^{r,i}\right)$ where $k_t^{r,1}, \cdots, k_t^{r,R}$ and $\beta_t^{r,1}, \cdots, \beta_t^{r,R}$ are two sets of R read keys and strengths received from the controller in the interface vector.

To achieve forward and backward iterations, we need to make the weightings go a step ahead or back from the location they recently read from. We can achieve that for the forward by multiplying the link matrix by the last read weightings. This shifts the weights from the last read location to the location where of the last write specified by the link matrix and constructs an intermediate forward weighting for each read head i: $\mathrm{f}_t^i = \mathrm{L}_t \mathrm{w}_{t-1}^{r,i}$. Similarly, we construct an intermediate backward weighting by multiplying the transpose of the link matrix by the last read weightings $\mathrm{b}_t^i = \mathrm{L}_{t-1}^{\top} \mathrm{w}_{t-1}^{r,i}$.

We can now construct the new read weightings for each read using the following rule:

$$\mathrm{w}_t^{r,i} = \pi_t^i[1]\mathrm{b}_t^i + \pi_t^i[2]\mathrm{c}_t^i + \pi_t^i[3]\mathrm{f}_t^i$$

where $\pi_t^1, \cdots, \pi_t^R$ are called the *read modes*. Each of these are a softmax distribution over three elements that come from the controller on the interface vector. Its three values determine the emphasis the read head should put on each read mechanism: backward, lookup, and forward, respectively. The controller learns to use these modes to instruct the memory on how data should be read.

The DNC Controller Network

Now that we've figured out the internal workings of the external memory in the DNC architecture, we're left with understanding how the controller that coordinates all the memory operations work. The controller's operation is simple: in its heart there's a neural network (recurrent or feed-forward) that takes in the input step along with the read-vectors from the last step and outputs a vector whose size depends on the architecture we chose for the network. Let's denote that vector by $\mathscr{N}(\chi_t)$, where $\mathscr{N}$ denotes whatever function is computed by the neural network, and χ_t denotes the concatenation of the input step and the last read vectors $\chi_t = \left[x_t; \mathrm{r}_{t-1}^1; \cdots; \mathrm{r}_{t-1}^R\right]$. This concate-

nation of the last read vectors serves a similar purpose as the hidden state in a regular LSTM: to condition the output on the past.

From that vector emitted from the neural network, we need two pieces of information. The first one is the interface vector ζ_t. As we saw, the interface vector holds all the information for the memory to carry out its operation. We can look at the ζ_t vector as a concatenation of the individual elements we encountered before, as depicted in Figure 8-7.

$$\zeta_t = [\underbrace{k_t^{r,1}; \ldots; k_t^{r,R}}_{\text{each of size } W}; \overbrace{\beta_t^{r,1}; \ldots; \beta_t^{r,R}}^{\text{each of size 1}}; k_t^w; \beta_t^w; \underbrace{e_t; v_t}_{\text{size } W}; \overbrace{f_t^1; \ldots; f_t^R; g_t^a; g_t^w}^{\text{each of size 1}}; \underbrace{\pi_t^1; \ldots; \pi_t^R}_{\text{each of size 3}}]$$

Figure 8-7. The interface vector decomposed to its individual components

By summing up the sizes along the components, we can consider the ζ_t vector as one big vector of size $R \times W + 3W + 5R + 3$. So in order to obtain that vector from the network output, we construct a learnable $|\mathcal{N}| \times (R \times W + 3W + 5R + 3)$ weights matrix W_ζ, where $|\mathcal{N}|$ is the size of the network's output, and such that:

$$\zeta_t = W_\zeta \mathcal{N}(\chi_t)$$

Before passing that ζ_t vector to the memory, we need to make sure that each component has a valid value. For example, all the gates as well as the erase vector must have values between 0 and 1, so we pass them through a sigmoid function to ensure that requirement:

$$e_t = \sigma(e_t), f_t^i = \sigma(f_t^i), g_t^a = \sigma(g_t^a), g_t^w = \sigma(g_t^w) \text{ where } \sigma(z) = \frac{1}{1 + e^{-z}}$$

Also, all the lookup strengths need to have a value larger than or equal to 1, so we pass them through a *oneplus* function first:

$$\beta_t^{r,i} = \text{oneplus}(\beta_t^{r,i}), \beta_t^w = \text{oneplus}(\beta_t^w) \text{ where } \text{oneplus}(z) = 1 + \log(1 + e^z)$$

And finally, the read modes must have a valid softmax distribution:

$$\pi_t^i = \text{softmax}(\pi_t^i) \text{ where } \text{softmax}(z) = \frac{e^z}{\Sigma_j e^{z_j}}$$

By these transformations, the interface vector is now ready to be passed to the memory; and while it guides the memory in its operations, we'll be needing a second piece of information from the neural network, the *pre-output* vector v_t. This is a vector of the same size of the final output vector, but it's not the final output vector. By using another learnable $|\mathcal{N}| \times Y$ weights matrix W_y, we can obtain the pre-output via:

$$v_t = W_y \mathcal{N}(\chi_t)$$

This pre-output vector gives us the ability to condition our final output not just on the network output, but also on the recently read vectors r_t from memory. Via a third learnable $(R \times W) \times Y$ weights matrix W_r, we can get the final output as:

$$;y_t = v_t + W_r\left[r_t^1; \cdots; r_t^R\right]$$

Given that the controller knows nothing about the memory except for the word size W, an already learned controller can be scaled to a larger memory with more locations without any need for retraining. Also, the fact that we didn't specify any particular structure for the neural network or any particular loss function makes DNC a universal architecture that can be applied to a variety of tasks and learning problems.

Visualizing the DNC in Action

One way to see DNC's operation in action is to train it on a simple task that would allow us to look at the weightings and the parameters' values and visualize them in an interpretable way. For this simple task, we'll use the copy problem we already saw with NTMs, but in a slightly modified form.

Instead of trying to copy a single sequence of binary vectors, our task here will be to copy a series of such sequences. Figure 8-8 (a) shows the single sequence input. After processing such single sequence input and copying the same sequence to the output, the DNC would have finished its program, and its memory would be reset in a way that will not allow us to see how it can dynamically manage it. Instead we'll treat a series of such sequences, shown in Figure 8-8 (b), as a single input.

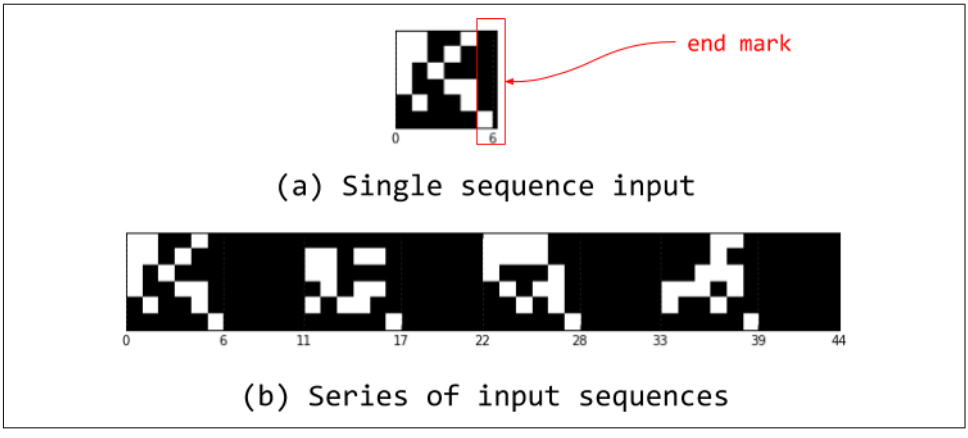


Figure 8-8. Single sequence input versus a series of input sequences

Figure 8-9 shows a visualization of the DNC operation after being trained on a series of length 4 where each sequence contains five binary vectors and an end mark. The DNC used here has only 10 memory locations, so there's no way it can store all 20

vectors in the input. A feed-forward controller is used to insure that nothing would be stored in a recurrent state, and only one read head is used to make the visualization more clear. These constraints should force the DNC to learn how to deallocate and reuse memory in order to successfully copy the whole input, and indeed it does.

We can see in that visualization how the DNC is writing each vector of the five in a sequence into a single memory location. As soon as the end mark is seen, the read head starts reading from these locations in the exact same order of writing. We can see how both the allocation and free gates alternate in activation between writing and reading phases of each sequence in the series. From the usage vector chart at the bottom, we can also see how after a memory location is written to, its usage becomes exactly 1, and how it drops to 0 just after reading from that location indicating that it was freed and can be reused again.

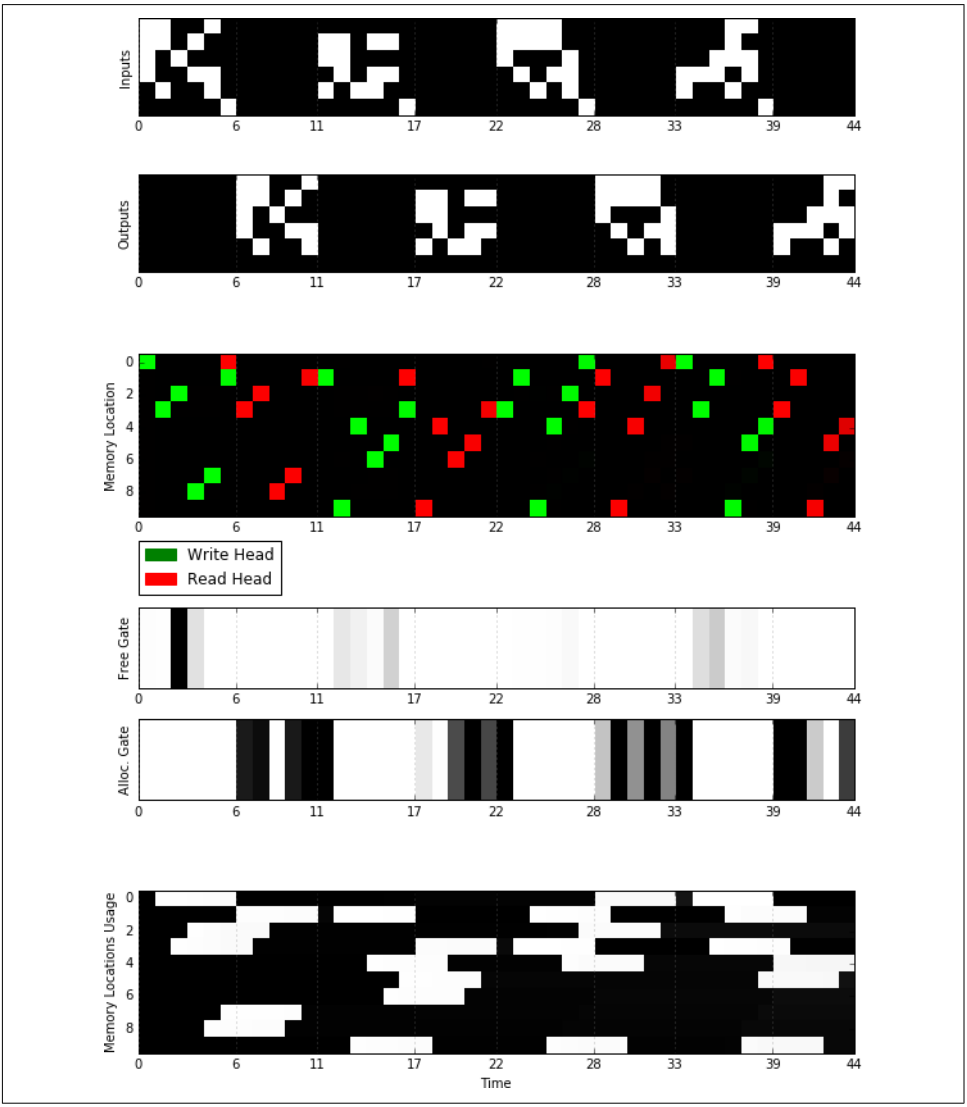


Figure 8-9. Visualization of the DNC operation on the copy problem

This visualization is part of the open source implementation of the DNC architecture by Mostafa Samir.[2] In the next section we'll learn the important tips and tricks that will allow us to implement a simpler version of DNC on the reading comprehension tasks.

2 *https://github.com/Mostafa-Samir/DNC-tensorflow*

Implementing the DNC in TensorFlow

Implementing the DNC architecture is essentially a direct application of the math we just discussed. So with the full implementation in the code repository associated with the book, we'll just be focusing on the tricky parts and introduce some new TensorFlow practice while we're at it.

The main part of the implementation resides in the *mem_ops.py* file where all of the attention and access mechanisms are implemented. This file is then imported to be used with the controller. Two operations that might be a little tricky to implement are the link matrix update and the allocation weighting calculation. Both of these operations can be naively implemented with `for` loops, but using `for` loops in creating a computational graph is generally not a good idea. Let's take the link matrix update operation first and see how it looks with a loop-based implementation:

```
def Lt(L, wwt, p, N):

    L_t = tf.zeros([N,N], tf.float32)
    for i in range(N):
        for j in range(N):
            if i == j:
                continue
            _mask = np.zeros([N,N], np.float32);
            _mask[i,j] = 1.0
            mask = tf.convert_to_tensor(_mask)

            link_t = (1 - wwt[i] - wwt[j]) * L[i,j] +
                      wwt[i] * p[j]
            L_t += mask * link_t

    return L_t
```

We used a masking trick here because TensorFlow doesn't support assignment for tensors' slices. We can find out what's wrong with this implementation by remembering that TensorFlow represents a type of programming called *symbolic*, where each call to an API doesn't carry out an operation and change the program state, but instead defines a node in a computational graph as a symbol for the operation we want to carry out. After that computational graph is fully defined, it's then fed with concrete values and executed. With that in mind, we can see, as depicted in Figure 8-10, how in most of the iterations of the `for` loop a new set of nodes representing the loop body gets added in the computational graph. So for N memory locations, we end up with $N^2 - N$ identical copies of the same nodes, each for each iteration, each taking up a chunk of our RAM and needing its own time to be processed before the next can be. When N is a small number, say 5, we get 20 identical copies, which is not so bad. However, if we want to use a larger memory, like with $N = 256$, we get 65,280 identical copies of the nodes, which is catastrophic for both the memory usage and the execution time!

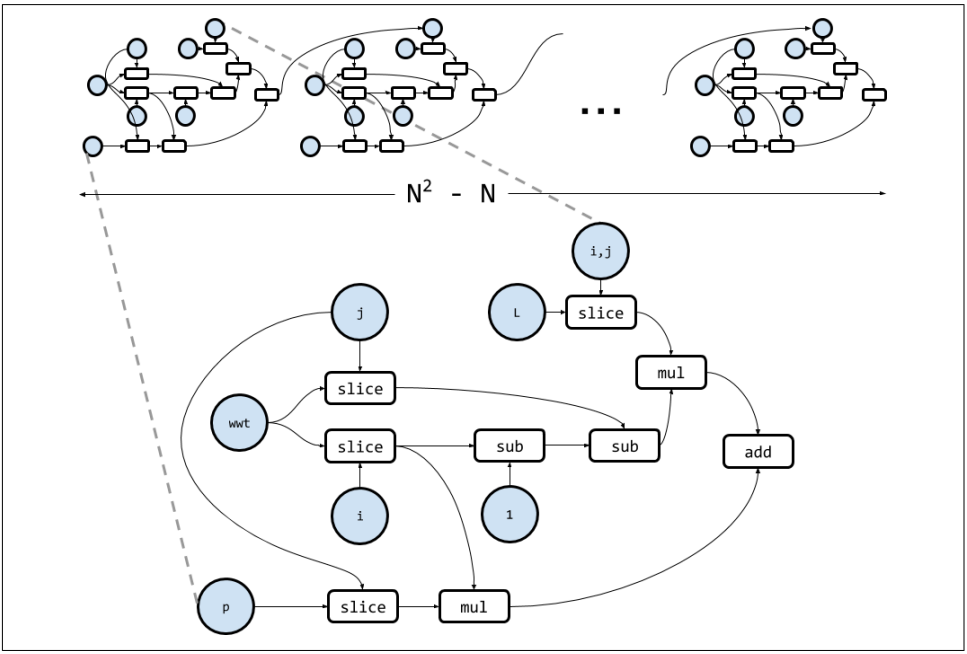


Figure 8-10. The computational graph of the link matrix update operation built with the for loop implementation

One possible way to overcome such issue is *vectorization*. In vectorization, we take an array operation that is originally defined in terms of individual elements and rewrite it as an operation on the whole array at once. For the link matrix update, we can rewrite the operation as:

$$\mathrm{L}_t = \left[\left(1 - \mathrm{w}_t^w \oplus \mathrm{w}_t^w\right) \circ \mathrm{L}_{t-1} + \mathrm{w}_t^w \mathrm{p}_{t-1}\right] \circ (1 - I)$$

Where I is the identity matrix, and the product $\mathrm{w}_t^w \mathrm{p}_{t-1}$ is an outer product. To achieve this vectorization, we define a new operator, the pairwise-addition of vectors, denoted by $\oplus$. This new operator is simply defined as:

$$u \oplus v = \begin{pmatrix} u_1 + v_1 & \cdots & u_1 + v_n \\ \vdots & \ddots & \vdots \\ u_n + v_1 & \cdots & u_n + v_n \end{pmatrix}$$

This operator adds a little bit to the memory requirements of the implementation, but not as much as the case in the loop-based implementation. With this vectorized reformulation of the update rule, we rewrite a more memory- and time-efficient implementation:

```
def Lt(L, wwt, p, N):

    # we only need the case of adding a single vector to itself
    def pairwise_add(v):
        n = v.get_shape().as_list()[0]
        # an NxN matrix of duplicates of u along the columns
        V = tf.concat(1, [v] * n)
        return V + V

    I = tf.constant(np.identity(N, dtype=np.float32))
    updated = (1 - pairwise_add(wwt)) * L + tf.matmul(wwt, p)
    updated = updated * (1 - I)  # eliminate self-links
    return updated
```

A similar process could be made for the allocation weightings rule. Instead of having a single rule for each element in the weighting vector, we can decompose it into a few operations that work on the whole vector at once:

1. While sorting the usage vector to get the free list, we also grab the sorted usage vector itself.
2. We calculate the cumulative product vector of the sorted usage. Each element of that vector is the same as the product term in our original element-wise rule.
3. We multiply the cumulative product vector by (1-the sorted usage vector). The resulting vector is the allocation weighting but in the sorted order, not the original order of the memory location.
4. For each element of that out-of-order allocation weighting, we take its value and put it in the corresponding index in the free list. The resulting vector is now the correct allocation weighting that we want.

Figure 8-11 summarizes this process with a numerical example.

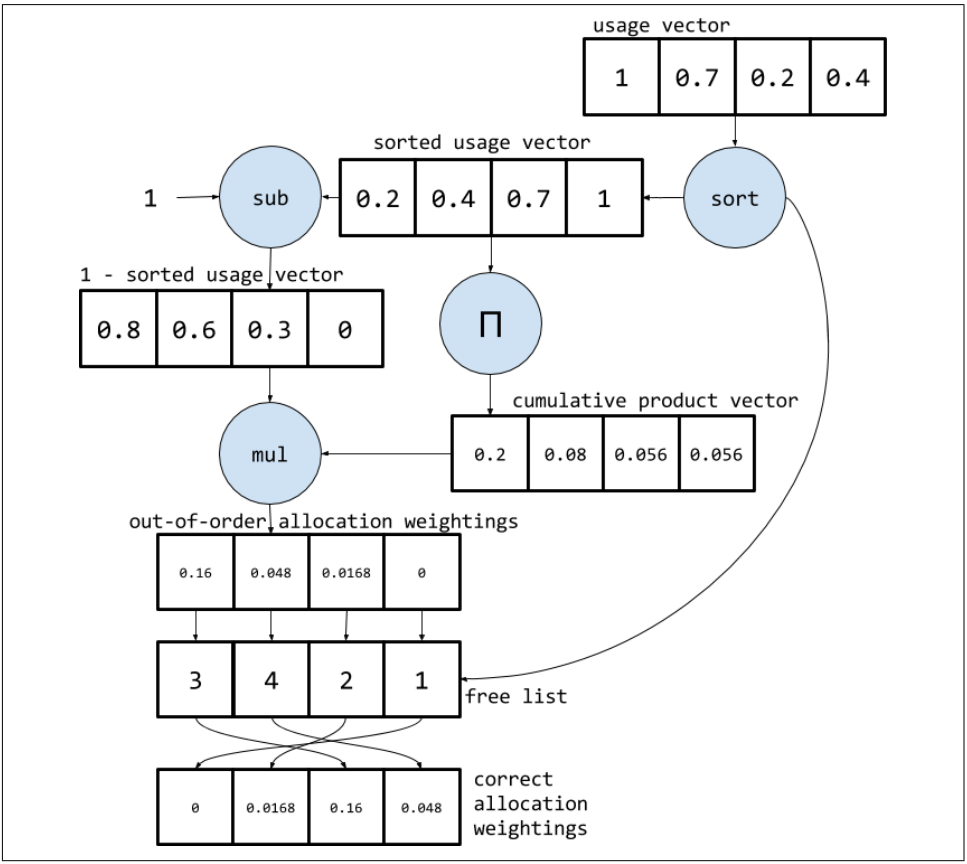


Figure 8-11. The vectorized process of calculating the allocation weightings

It may seem that we still need loops for the sorting operation in step 1 and for reordering the weights in step 4, but fortunately TensorFlow provides symbolic operations that would allow us to carry out these operations without the need for a Python loop.

For sorting we'll be using `tf.nn.top_k`. This operation takes a tensor and a number k and returns both the sorted top k values in descending order and the indices of these values. To get the sorted usage vector in ascending order, we need to get the top N values of the negative of the usage vector. We can bring back the sorted values to their original signs by multiplying the resulting vector by -1:

```
sorted_ut, free_list = tf.nn.top_k(-1 * ut, N)
sorted_ut *= -1
```

For reordering the allocation weights, we'll make use of a new TensorFlow data structure called `TensorArray`. We can think of these tensor arrays as a symbolic alterna-

tive for Python's list. We first create an empty tensor array of size N to be the container of the weights in their correct order, and then put the values at their correct places using the instance method `scatter(indices, values)`. This method takes in its second argument a tensor and scatters the values along its first dimension across the array, with the first argument being a list of indices of the locations to which we want to scatter the corresponding values. In our case here, the first argument is the free list, and the second is the out-of-order allocation weightings. Once we get the array with the weights in the correct places, we use another instance method `pack()` to wrap up the whole array into a `Tensor` object:

```
empty_at = tf.TensorArray(tf.float32, N)
full_at = empty_at.scatter(free_list, out_of_location_at)

a_t = full_at.pack()
```

The last part of the implementation that requires looping is the controller loop itself, the loop that goes over each step of the input sequence to process it. Because vectorization only works when operations are defined element-wise, the controller's loop can't be vectorized. Fortunately, TensorFlow still provides us with a method to escape Python's `for` loops and their massive performance hit; this method is the *symbolic loop.* Symbolic loops work like most of our symbolic operations: instead of unrolling the actual loop into the graph, it defines a node that would be executed as a loop when the graph is executed.

We can define a symbolic loop using `tf.while_loop(cond, body, loop_vars)`. The `loop_vars` argument is a list of the initial values of tensors and/or tensor arrays that are passed through each iteration of the loop; this list can possibly be nested. The other two arguments are callables (functions or lambdas) that are passed to this list of loop variables at each iteration. The first argument `cond` represents the loop condition. As long as this callable is returning true, the loop will keep on working. The other argument `body` represents the body of the loop that gets executed at each iteration. This callable is the one responsible for modifying the loop variables and returning them back to the next iteration. Such modifications, however, must keep the tensor's shape consistent throughout the iterations. After the loop is executed, the list of loop variables with their values after the last iteration is returned.

To get a better understanding of how symbolic loops can be used, we'll try now to apply this to a simple use case. Suppose that we are given a vector of values and we want to get its cumulative sum vector. We achieve that with `tf.while_loop`, as in the following code:

```
values = tf.random_normal([10])

index = tf.constant(0)
values_array = tf.TensorArray(tf.float32, 10)
cumsum_value = tf.constant(0.)
cumsum_array = tf.TensorArray(tf.float32, 10)

values_array = values_array.unpack(values)

def loop_body(index, values_array, cumsum_value, cumsum_array):
    current_value = values_array.read(index)
    cumsum_value += current_value
    cumsum_array = cumsum_array.write(index, cumsum_value)
    index += 1

    return (index, values_array, cumsum_value, cumsum_array)

_, _, _, final_cumsum = tf.while_loop(
    cond= lambda index, *_: index < 10,
    body= loop_body,
    loop_vars= (index, values_array, cumsum_value,
                cumsum_array)
)

cumsum_vector = final_cumsum.pack()
```

We first use the `unpack(values)` of the tensor array to unpack a tensor's values along its first dimension across the array. In the body loop we get the value at the current index using the `read(index)` method, which returns the value at the given index in the array. We then calculate the cumulative sum so far and add it to the cumulative sum array using the `write(index, value)` method which writes the given value in the array at the given index. Finally, after the loop is fully executed, we get the final cumulative sum array and pack it into a tensor. A similar pattern is used to implement the DNC's loop over the input sequence steps.

Teaching a DNC to Read and Comprehend

Earlier in the chapter, back when we were talking about neural n-grams, we said that it's not of the complexity of an AI that can answer questions after reading a story. Now we have reached the point that we can build such a system because this is exactly what DNCs do when applied on the bAbI dataset.

The bAbI dataset is a synthetic dataset consisting of 20 sets of stories, questions on those stories, and their answers. Each set represents a specific and unique task of reasoning and inference from text. In the version we'll use, each task contains 10,000 questions for training and 1,000 questions for testing. For example, the following story (from which the passage we saw earlier was adapted) is from the *lists-and-sets*

task where the answers to the questions are lists/sets of objects mentioned in the story:

```
1 Mary took the milk there.
2 Mary went to the office.
3 What is Mary carrying?   milk 1
4 Mary took the apple there.
5 Sandra journeyed to the bedroom.
6 What is Mary carrying?   milk,apple 1 4
```

This is taken directly from the dataset, and as you can see, a story is organized into numbered sentences that start from 1. Each question ends with a question mark, and the words that directly follow the question mark are the answers. If an answer consists of more than one word, the words are separated by commas. The numbers that follow the answers are supervisory signals that point to the sentences that contain the answers' words.

To make the tasks more challenging, we'll discard these supervisory signals and let the system learn to read the text and figure out the answer on its own. Following the DNC paper, we'll preprocess our dataset by removing all the numbers and punctuation except for "?" and ".", bringing all the words to lowercase, and replacing the answer words with dashes "-" in the input sequence. After this we get 159 unique words and marks (lexicons) across all the tasks, so we'll encode each lexicon as a one-hot vector of size 159, no embeddings, just the plain words directly. Finally, we combine all the of 200,000 training questions to train the model jointly on them, and we keep each task's test questions separate to test the trained model afterward on each task individually. This whole process is implemented in the *preprocess.py* file in the code repository.

To train the model, we randomly sample a story from the encoded training data, pass it through the DNC with an LSTM controller, and get the corresponding output sequence. We then measure the loss between the output sequence and the desired sequence using the softmax cross-entropy loss, but only on the steps that contain answers. All the other steps are ignored by weighting the loss with a weights vector that has 1 at the answer's steps and 0 elsewhere. This process is implemented in the *train_babi.py* file.

After the model is trained, we test its performance on the remaining test questions. Our metric will be the percentage of questions the model failed to answer in each task. An answer to a question is the word with the largest softmax value in the output, or the most probable word. A question is considered to be answered correctly if all of its answer's words are the correct words. If the model failed to answer more than 5% of a task's questions, we consider that the model failed on that task. The testing procedure is found in the *test_babi.py* file.

After training the model for about 500,000 iterations (caution, it takes a long time!), we can see that it's performing pretty well on most of the tasks. At the same time, it's

performing badly on more difficult tasks like *path-finding*, where the task is to answer questions about how to get from one place to another. The following report compares our model's results to the mean values reported in the original DNC paper:

```
Task                        Result          Paper's Mean
-----------------------------------------------------------
single supporting fact      0.00%           9.0±12.6%
two supporting facts        11.88%          39.2±20.5%
three supporting facts      27.80%          39.6±16.4%
two arg relations           1.40%           0.4±0.7%
three arg relations         1.70%           1.5±1.0%
yes no questions            0.50%           6.9±7.5%
counting                    4.90%           9.8±7.0%
lists sets                  2.10%           5.5±5.9%
simple negation             0.80%           7.7±8.3%
indefinite knowledge        1.70%           9.6±11.4%
basic coreference           0.10%           3.3±5.7%
conjunction                 0.00%           5.0±6.3%
compound coreference        0.40%           3.1±3.6%
time reasoning              11.80%          11.0±7.5%
basic deduction             45.44%          27.2±20.1%
basic induction             56.43%          53.6±1.9%
positional reasoning        39.02%          32.4±8.0%
size reasoning              8.68%           4.2±1.8%
path finding                98.21%          64.6±37.4%
agents motivations          2.71%           0.0±0.1%
-----------------------------------------------------------
Mean Err.                   15.78%          16.7±7.6%
Failed (err. > 5%)          8               11.2±5.4
```

Summary

In this chapter, we've explored the cutting edge of deep learning research with NTMs and DNCs, culminating with the implementation of a model that can solve an involved reading comprehension task.

In the final chapter of this book, we'll begin to explore a very different space of problems known as reinforcement learning. We'll build an intuition for this new class of tasks and develop an algorithmic foundation to tackle these problems using the deep learning tools we've developed thus far.

CHAPTER 9

Deep Reinforcement Learning

Nicholas Locascio[1]

In this chapter, we'll discuss reinforcement learning, which is a branch of machine learning that deals with learning via interaction and feedback. Reinforcement learning is essential to building agents that can not only perceive and interpret the world, but also take action and interact with it. We will discuss how to incorporate deep neural networks into the framework of reinforcement learning and discuss recent advances and improvements in this field.

Deep Reinforcement Learning Masters Atari Games

The application of deep neural networks to reinforcement learning had a major breakthrough in 2014, when the London startup DeepMind astonished the machine learning community by unveiling a deep neural network that could learn to play Atari games with superhuman skill. This network, termed a *Deep Q-Network* (DQN) was the first large-scale successful application of reinforcement learning with deep neural networks. DQN was so remarkable because the same architecture, without any changes, was capable of learning 49 different Atari games, despite each game having different rules, goals, and gameplay structure. To accomplish this feat, DeepMind brought together many traditional ideas in reinforcement learning while also developing a few novel techniques that proved key to DQN's success. Later in this chapter we will implement DQN, as it is described in the *Nature* paper "Human-level control through deep reinforcement learning."[2] But first, let's take a dive into reinforcement learning (Figure 9-1).

---

1 *http://nicklocascio.com/*

2 Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.

Figure 9-1. A deep reinforcement learning agent playing Breakout. This image is from the OpenAI Gym[3] DQN agent that we build in this chapter.

3 Brockman, Greg, et al. "OpenAI Gym." *arXiv preprint arXiv*:1606.01540 (2016). *https://gym.openai.com/*

What Is Reinforcement Learning?

Reinforcement learning, at its essentials, is learning by interacting with an environment. This learning process involves an *actor*, an *environment*, and a *reward signal*. The actor chooses to take an action in the environment, for which the actor is rewarded accordingly. The way in which an actor chooses actions is called a *policy*. The actor wants to increase the reward it receives, and so must learn an optimal policy for interacting with the environment (Figure 9-2).

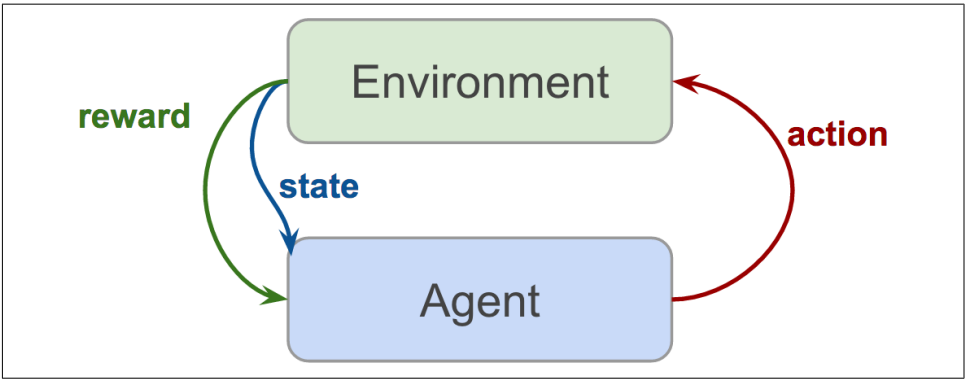


Figure 9-2. Reinforcement learning setup

Reinforcement learning is different from the other types of learning that we have covered thus far. In traditional supervised learning, we are given data and labels, and are tasked with predicting labels given data. In unsupervised learning, we are given just data and are tasked with discovering underlying structure in this data. In reinforcement learning, we are given neither data nor labels. Our learning signal is derived from the rewards given to the agent by the environment.

Reinforcement learning is exciting to many in the artificial intelligence community because it is a general-purpose framework for creating intelligent agents. Given an environment and some rewards, the agent learns to interact with that environment to maximize its total reward. This type of learning is more in line with how humans develop. Yes, we can build a pretty good model to classify dogs from cats with extremely high accuracy by training on thousands of images. But you won't find this approach used in any elementary schools. Humans interact with their environment to learn representations of the world which they can use to make decisions.

Furthermore, reinforcement learning applications are at the forefront of many cutting-edge technologies including self-driving cars, robotic motor control, game playing, air-conditioning control, ad-placement optimization, and stock market trading strategies.

As an illustrative exercise, we'll be tackling a simple reinforcement learning and control problem called pole-balancing. In this problem, there is a cart with a pole that is connected by a hinge, so the pole can swing around the cart. There is an agent that can control the cart, moving it left or right. There is an environment, which rewards the agent when the pole is pointed upward, and penalizes the agent when the pole falls over (Figure 9-3).

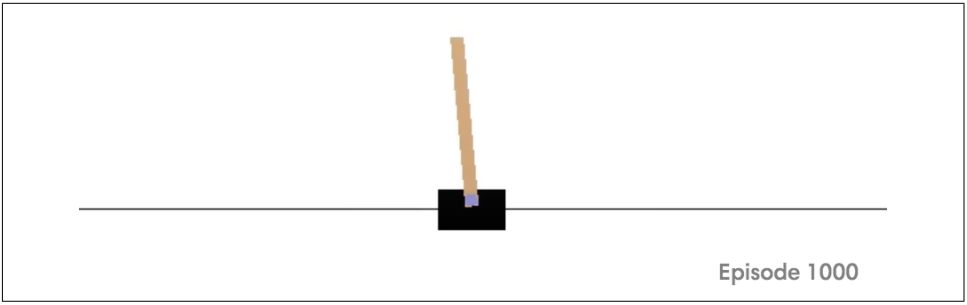


Figure 9-3. A simple reinforcement learning agent balancing a pole. This image is from our OpenAI Gym Policy Gradient agent that we build in this chapter.

Markov Decision Processes (MDP)

Our pole-balancing example has a few important elements, which we formalize as a *Markov Decision Process* (MDP). These elements are:

State
: The cart has a range of possible places on the x-plane where it can be. Similarly, the pole has a range of possible angles.

Action
: The agent can take action by moving the cart either left or right.

State Transition
: When the agent acts, the environment changes—the cart moves and the pole changes angle and velocity.

Reward
: If an agent balances the pole well, it receives a positive reward. If the pole falls, the agent receives a negative reward.

An MDP is defined as the following:

- S, a finite set of possible states
- A, a finite set of actions
- $P(r, s' \mid s, a)$, a state transition function
- R, reward function

MDPs offer a mathematical framework for modeling decision-making in a given environment (Figure 9-4).

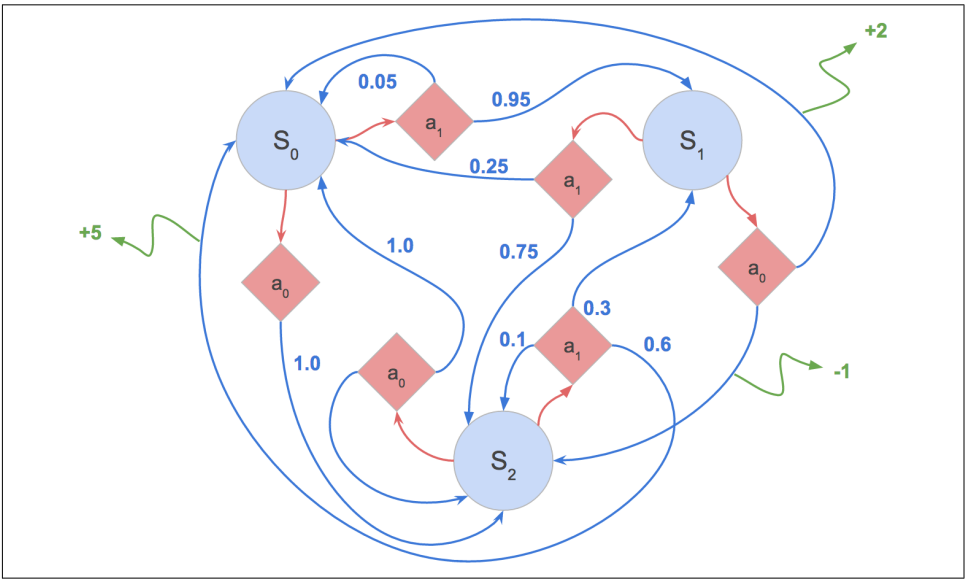


Figure 9-4. An example of an MDP. Blue circles represent the states of the environment. Red diamonds represent actions that can be taken. The edges from diamonds to circles represent the transition from one state to the next. The numbers along these edges represent the probability of taking a certain action. The numbers at the end of the green arrows represent the reward given to the agent for making the given transition.

As an agent takes action in an MDP framework, it forms an *episode*. An episode consists of series of tuples of states, actions, and rewards. Episodes run until the environment reaches a terminal state, like the "Game Over" screen in Atari games, or when the pole hits the ground in our pole-cart example. The following equation shows the variables in an episode:

$$(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots (s_n, a_n, r_n)$$

In pole-cart, our environment state can be a tuple of the position of the cart and the angle of the pole, like so: $(x_{cart}, \theta_{pole})$.

Policy

MDP's aim is to find an optimal policy for our agent. *Policies* are the way in which our agent acts based on its current state. Formally, policies can be represented as a function π that chooses the action a that the agent will take in state s.

The objective of our MDP is to find a policy to maximize the expected future return:

$$\max_{\pi} \; E\left[R_0 + R_1 + \ldots R_t \middle| \pi\right]$$

In this objective, R represents the *future return* of each episode. Let's define exactly what future return means.

Future Return

Future return is how we consider the rewards of the future. Choosing the best action requires consideration of not only the immediate effects of that action, but also the long-term consequences. Sometimes the best action actually has a negative immediate effect, but a better long-term result. For example, a mountain-climbing agent that is rewarded by its altitude may actually have to climb downhill to reach a better path to the mountain's peak.

Therefore, we want our agents to optimize for *future return*. In order to do that, the agent must consider the future consequences of its actions. For example, in a game of Pong, the agent receives a reward when the ball passes into the opponent's goal. However, the actions responsible for this reward (the inputs that position the racquet to strike scoring hit) happen many time steps before the reward is received. The reward for each of those actions is delayed.

We can incorporate delayed rewards into our overall reward signal by constructing a *return* for each time step that takes into account future rewards as well as immediate rewards. A naive approach for calculating *future return* for a time step may be a simple sum like so:

$$R_t = \Sigma_{k=0}^{T} r_{t+k}$$

We can calculate all returns, R, where $R = \{R_0, R_1, \ldots R_i, \ldots R_n\}$ with the following code:

```
def calculate_naive_returns(rewards):
""" Calculates a list of naive returns given a
    list of rewards."""
    total_returns = np.zeros(len(rewards))
    total_return = 0.0
    for t in range(len(rewards), 0):
        total_return = total_return + reward
        total_returns[t] = total_return
    return total_returns
```

This naive approach successfully incorporates future rewards so the agent can learn an optimal global policy. This approach values future rewards equally to immediate rewards. However, this equal consideration of all rewards is problematic. With infinite time steps, this expression can diverge to infinity, so we must to find a way to bound it. Furthermore, with equal consideration at each time step, the agent can optimize for a very future reward, and we would learn a policy that lacks any sense of urgency or time sensitivity in pursuing its rewards.

Instead, we should value future rewards slightly less in order to force our agents to learn to get rewards quickly. We accomplish this with a strategy called *discounted future return*.

Discounted Future Return

To implement discounted future return, we scale the reward of a current state by the discount factor, γ, to the power of the current time step. In this way, we penalize agents that take many actions before receiving positive reward. Discounted rewards bias our agent to prefer receiving reward in immediate future, which is advantageous to learning a good policy. We can express the reward as follows:

$$R_t = \Sigma_{k=0}^{T} \gamma^t r_{t+k+1}$$

The discount factor, γ, represents the level of discounting we want to achieve and can be between 0 and 1. High γ means little discounting, low γ provides much discounting. A typical γ hyperparameter setting is between 0.99 and 0.97.

We can implement discounted return like so:

```
def discount_rewards(rewards, gamma=0.98):
    discounted_returns = [0 for _ in rewards]
    discounted_returns[-1] = rewards[-1]
    for t in range(len(rewards)-2, -1, -1): # iterate backwards
        discounted_returns[t] = rewards[t] +
          discounted_returns[t+1]*gamma
    return discounted_returns
```

Explore Versus Exploit

Reinforcement learning is fundamentally a trial-and-error process. In such a framework, an agent afraid to make mistakes can prove to be highly problematic. Consider the following scenario. A mouse is placed in the maze shown in Figure 9-5. Our agent must control the mouse to maximize reward. If the mouse gets the water, it receives a reward of +1; if the mouse reaches a poison container (red), it receives a reward of -10; if the mouse gets the cheese, it receives a reward of +100. Upon receiving reward, the episode is over. The optimal policy involves the mouse successfully navigating to the cheese and eating it.

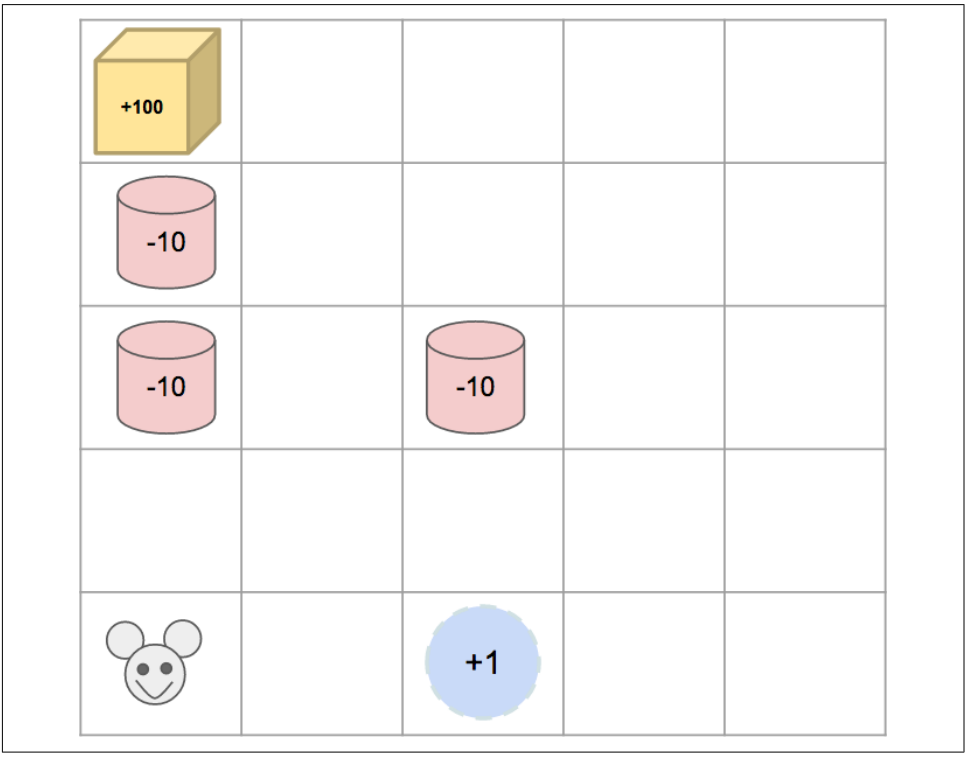


Figure 9-5. A predicament that many mice find themselves in

In the first episode, the mouse takes the left route, steps on a trap, and receives a -10 reward. In the second episode, the mouse avoids the left path, since it resulted in such a negative reward, and drinks the water immediately to its right for a +1 reward. After two episodes, it would seem that the mouse has found a good policy. It continues to follow its learned policy on subsequent episodes and achieves the moderate +1 reward reliably. Since our agent utilizes a greedy strategy—always choosing the model's best action—it is stuck in a policy that is a *local maximum*.

To prevent such a situation, it may be useful for the agent to deviate from the model's recommendation and take a suboptimal action in order to *explore* more of the environment. So instead of taking the immediate right turn to *exploit* the environment to get water and the reliable +1 reward, our agent may choose to take a left turn and venture into more treacherous areas in search of a more optimal policy. Too much exploration, and our agent fails to optimize any reward. Not enough exploration can result in our agent getting stuck in local minimum. This balance of *explore versus exploit* is crucial to learning a successful policy.

ϵ-Greedy

One strategy for balancing the explore-exploit dilemma is called *e-Greedy*. e-Greedy is a simple strategy that involves making a choice at each step to either take the agent's top recommended action or take a random action. The probability that the agent takes a random action is the value known as ϵ:

We can implement ϵ-Greedy like so:

```
def epsilon_greedy_action(action_distribution, epsilon=1e-1):
    if random.random() < epsilon:
        return np.argmax(np.random.random(
           action_distribution.shape))
    else:
        return np.argmax(action_distribution)
```

Annealed ϵ-Greedy

When training a reinforcement learning model, oftentimes we want to do more exploring in the beginning since our model knows little of the world. Later, once our model has seen much of the environment and learned a good policy, we want our agent to trust itself more to further optimize its policy. To accomplish this, we cast aside the idea of a fixed ϵ, and instead anneal it over time, having it start low and increase by a factor after each training episode. Typical settings for annealed e-Greedy scenarios include annealing from 0.99 to 0.1 over 10,000 scenarios. We can implement annealing like so:

```
def epsilon_greedy_action_annealed(action_distribution,
                                   percentage,
                                   epsilon_start=1.0,
                                   epsilon_end=1e-2):
    annealed_epsilon = epsilon_start*(1.0-percentage) +
      epsilon_end*percentage
    if random.random() < annealed_epsilon:
        return np.argmax(np.random.random(
          action_distribution.shape))
    else:
        return np.argmax(action_distribution)
```

Policy Versus Value Learning

So far we've defined the setup of reinforcement learning, discussed discounted future return, and looked at the trade-offs of explore versus exploit. What we haven't talked about is how we're actually going to teach an agent to maximize its reward. Approaches to this fall into two broad categories: *policy learning* and *value learning*. In policy learning, we are directly learning a policy that maximizes reward. In value learning, we are learning the value of every state + action pair. If you were trying to learn to ride a bike, a policy learning approach would be to think about how

pushing on the right pedal while you were falling to the left would course-correct you. If you were trying to learn to ride a bike with a value learning approach, you would assign a score to different bike orientations and actions you can take in those positions. We'll be covering both in this chapter, so let's start with policy learning.

Policy Learning via Policy Gradients

In typical supervised learning, we can use stochastic gradient descent to update our parameters to minimize the loss computed from our network's output and the true label. We are optimizing the expression:

$$\arg\ \min_\theta \Sigma_i \log p(y_i \mid x_i; \theta)$$

In reinforcement learning, we don't have a true label, only reward signals. However, we can still use SGD to optimize our weights using something called *policy gradients*.[4] We can use the actions the agent takes, and the returns associated with those actions, to encourage our model weights to take good actions that lead to high reward, and to avoid bad ones that lead to low reward. The expression we optimize for is:

$$\arg\ \min_\theta\ -\Sigma_i R_i \log p(y_i \mid x_i; \theta)$$

where y_i is the action taken by the agent at time step t and where R_i is our discounted future return**.** A In this way, we scale our loss by the value of our return, so if the model chose an action that led to negative return, this would lead to greater loss. Furthermore, if the model is very confident in that bad decision, it would get penalized even more, since we are taking into account the log probability of the model choosing that action. With our loss function defined, we can apply SGD to minimize our loss and learn a good policy.

Pole-Cart with Policy Gradients

We're going to implement a policy-gradient agent to solve pole-cart, a classic reinforcement learning problem. We will be using an environment from the OpenAi Gym created just for this task.

OpenAI Gym

The OpenAI Gym is a Python toolkit for developing reinforcement agents. OpenAI Gym provides an easy-to-use interface for interacting with a variety of environments. It contains over 100 open-source implementations of common reinforcement learning environments. OpenAI Gym speeds up development of reinforcement learning

4 Sutton, Richard S., et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." NIPS. Vol. 99. 1999.

agents by handling everything on the environment simulation side, allowing researchers to focus on their agent and learning algorithms. Another benefit of OpenAI Gym is that researchers can fairly compare and evaluate their results with others because they can all use the same standardized environment for a task. We'll be using the pole-cart environment from OpenAI Gym to create an agent that can easily interact with this environment.

Creating an Agent

To create an agent that can interact with an OpenAI environment, we'll define a class PGAgent, which will contain our model architecture, model weights, and hyperparameters:

```
class PGAgent(object):

    def __init__(self, session, state_size, num_actions,
        hidden_size, learning_rate=1e-3,
        explore_exploit_setting=
          'epsilon_greedy_annealed_1.0->0.001'):
        self.session = session
        self.state_size = state_size
        self.num_actions = num_actions
        self.hidden_size = hidden_size
        self.learning_rate = learning_rate
        self.explore_exploit_setting = explore_exploit_setting

        self.build_model()
        self.build_training()

    def build_model(self):
        with tf.variable_scope('pg-model'):
            self.state = tf.placeholder(
                shape=[None, self.state_size],
                dtype=tf.float32)
            self.h0 = slim.fully_connected(self.state,
            self.hidden_size)
            self.h1 = slim.fully_connected(self.h0,
            self.hidden_size)
            self.output = slim.fully_connected(
                self.h1, self.num_actions,
                activation_fn=tf.nn.softmax)
            # self.output = slim.fully_connected(self.h1,
              self.num_actions)

    def build_training(self):
        self.action_input = tf.placeholder(tf.int32,
          shape=[None])
        self.reward_input = tf.placeholder(tf.float32,
          shape=[None])
```

```
    # Select the logits related to the action taken
    self.output_index_for_actions = (tf.range(
        0, tf.shape(self.output)[0]) *
          tf.shape(self.output)[1]) +
           self.action_input
    self.logits_for_actions = tf.gather(
        tf.reshape(self.output, [-1]),
        self.output_index_for_actions)

    self.loss = - \
        tf.reduce_mean(tf.log(self.logits_for_actions) *
          self.reward_input)

    self.optimizer = tf.train.AdamOptimizer(
        learning_rate=self.learning_rate)
    self.train_step = self.optimizer.minimize(self.loss)

def sample_action_from_distribution(
  self, action_distribution,
  epsilon_percentage):
    # Choose an action based on the action probability
    # distribution and an explore vs exploit
    if self.explore_exploit_setting == 'greedy':
        action = greedy_action(action_distribution)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_0.05':
        action = epsilon_greedy_action(action_distribution,
          0.05)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_0.25':
        action = epsilon_greedy_action(action_distribution,
          0.25)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_0.50':
        action = epsilon_greedy_action(action_distribution,
          0.50)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_0.90':
        action = epsilon_greedy_action(action_distribution,
          0.90)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_annealed_1.0->0.001':
        action = epsilon_greedy_action_annealed(
            action_distribution, epsilon_percentage, 1.0,
              0.001)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_annealed_0.5->0.001':
        action = epsilon_greedy_action_annealed(
            action_distribution, epsilon_percentage, 0.5,
              0.001)
    elif self.explore_exploit_setting ==
      'epsilon_greedy_annealed_0.25->0.001':
```

```
            action = epsilon_greedy_action_annealed(
                action_distribution, epsilon_percentage, 0.25,
                  0.001)

        return action

    def predict_action(self, state, epsilon_percentage):
        action_distribution = self.session.run(
            self.output, feed_dict={self.state: [state]})[0]
        action = self.sample_action_from_distribution(
            action_distribution, epsilon_percentage)
        return action
```

Building the Model and Optimizer

Lets break down some important functions. In `build_model()`, we define our model architecture as a three-layer neural network. The model returns a layer of three nodes, each representing the model's action probability distribution. In `build_training()`, we implement our policy gradient optimizer. We express our objective loss as we talked about, scaling the model's prediction probability for an action with the return received for taking that action, and summing these all up to form a minibatch. With our objective defined, we can use `tf.AdamOptimizer`, which will adjust our weights according to the gradient to minimize our loss.

Sampling Actions

We define the `predict_action` function, which samples an action based on the model's action probability distribution output. We support the various sampling strategies that we talked about to balance explore versus exploit, including greedy, epsilon greedy, and epsilon greedy annealing.

Keeping Track of History

We'll be aggregating our gradients from multiple episode runs, so it will be useful to keep track of state, action, and reward tuples. To this end, we implement an episode history and memory:

```
class EpisodeHistory(object):

    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

    def add_to_history(self, state, action, reward,
      state_prime):
```

```
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)
        self.state_primes.append(state_prime)


class Memory(object):

    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

    def reset_memory(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

    def add_episode(self, episode):
        self.states += episode.states
        self.actions += episode.actions
        self.rewards += episode.rewards
        self.discounted_returns += episode.discounted_returns
```

Policy Gradient Main Function

Lets put this all together in our main function, which will create an OpenAI Gym environment for CartPole, make an instance of our agent, and have our agent interact with and train on the CartPole environment:

```
def main(argv):
    # Configure Settings
    total_episodes = 5000
    total_steps_max = 10000
    epsilon_stop = 3000
    train_frequency = 8
    max_episode_length = 500
    render_start = -1
    should_render = False

    explore_exploit_setting =
      'epsilon_greedy_annealed_1.0->0.001'

    env = gym.make('CartPole-v0')
    state_size = env.observation_space.shape[0]  # 4 for
                                                 # CartPole-v0
    num_actions = env.action_space.n  # 2 for CartPole-v0
```

```
solved = False
with tf.Session() as session:
    agent = PGAgent(session=session, state_size=state_size,
      num_actions=num_actions,
                    hidden_size=16,
                      explore_exploit_setting=
                        explore_exploit_setting)
    session.run(tf.global_variables_initializer())

    episode_rewards = []
    batch_losses = []

    global_memory = Memory()
    steps = 0
    for i in tqdm.tqdm(range(total_episodes)):
        state = env.reset()
        episode_reward = 0.0
        episode_history = EpisodeHistory()
        epsilon_percentage = float(min(i/float(
          epsilon_stop), 1.0))
        for j in range(max_episode_length):
            action = agent.predict_action(state,
              epsilon_percentage)

            state_prime, reward, terminal, _ =
              env.step(action)
            if (render_start > 0 and i >
              render_start and should_render) \
                or (solved and should_render):
                env.render()
            episode_history.add_to_history(
                state, action, reward, state_prime)
            state = state_prime
            episode_reward += reward
            steps += 1
            if terminal:
                episode_history.discounted_returns =
                  discount_rewards(
                    episode_history.rewards)
                global_memory.add_episode(
                  episode_history)

                if np.mod(i, train_frequency) == 0:
                    feed_dict = {
                    agent.reward_input: np.array(
                      global_memory.discounted_returns),
                    agent.action_input: np.array(
                      global_memory.actions),
                    agent.state: np.array(
                      global_memory.states)}
                    _, batch_loss = session.run(
```

```
                    [agent.train_step, agent.loss],
                      feed_dict=feed_dict)
                batch_losses.append(batch_loss)
                global_memory.reset_memory()

            episode_rewards.append(episode_reward)
            break

    if i % 10:
        if np.mean(episode_rewards[:-100]) >
          100.0:
            solved = True
        else:
            solved = False
```

This code will train a CartPole agent to successfully and consistently balance the pole.

PGAgent Performance on Pole-Cart

Figure 9-6 is a chart of the average reward of our agent at each step of training. We try out 8 different sampling methods, and achieve best results with epsilon greedy annealing from 1.0 to 0.001.

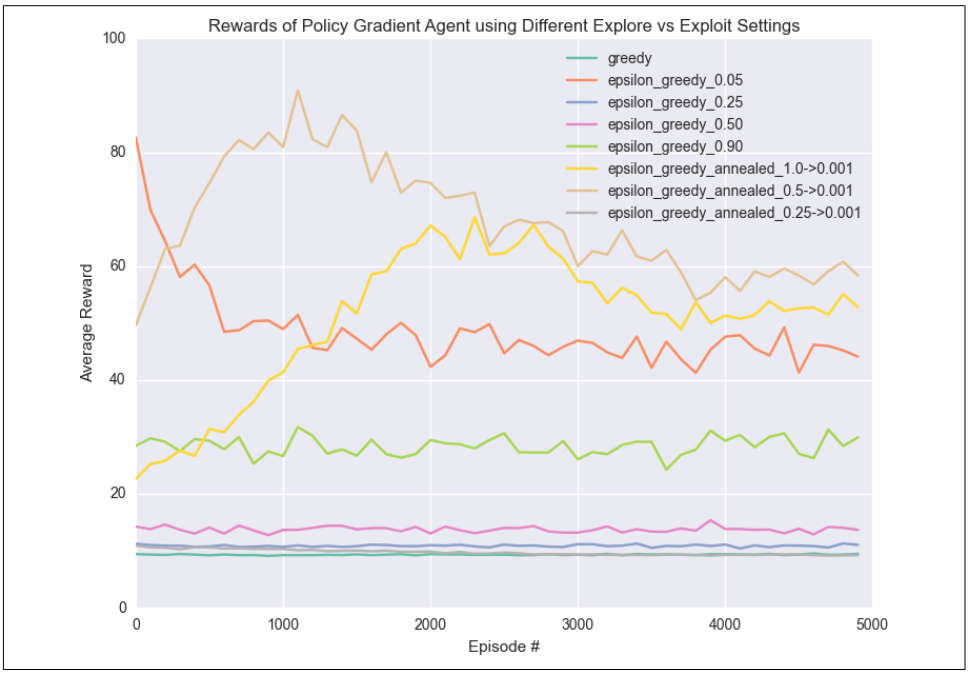


Figure 9-6. Explore-exploit configurations affect how fast and how well learning occurs

Notice how, across the board, standard epsilon greedy does very poorly. Lets talk about why this might be. With a high epsilon set to 0.9, we are taking a random action 90% of the time. Even if the model learns to execute the perfect actions, we'll still only be using these 10% of the time. On the other end, with a low epsilon of 0.05, we are taking what our model believes to be optimal actions the vast majority of the time. This performance is a bit better, but gets stuck in a local reward minimum because it lacks the ability to explore other strategies. So neither epsilon greedy of 0.05 nor 0.9 gives us great results. The former places too much emphasis on exploration, and the latter, too little. This is why epsilon annealing is such a powerful sampling strategy. It allows the model to explore early and exploit late, which is crucial to learning good policies.

Q-Learning and Deep Q-Networks

Q-learning is in the category of reinforcement learning called value-learning. Instead of directly learning a policy, we will be learning the value of states and actions. Q-learning involves learning a function, a *Q-function*, which represents the quality of a state, action pair. The Q-function, defined *Q(s, a)*, is a function that calculates the maximum discounted future return when action a is performed in state s.

The Q-value represents our expected long-term rewards, given we are at a state, and take an action, and then take every subsequent action perfectly (to maximize expected future reward). This can be expressed formally as:

$$Q^*(s_t, a_t) = max_\pi E\left[\Sigma_{i=t}^{T} \gamma^i r^i\right]$$

A question you may be asking is, how can we know Q-values? It is difficult, even for humans, to know how good an action is, because you need to know how you are going to act in the future. Our expected future returns depend on what our long-term strategy is going to be. This seems to be a bit of a chicken-and-egg problem. In order to value a state, action pair you need to know all the perfect subsequent actions. And in order to know the best actions, you need to have accurate values for a state and action.

The Bellman Equation

We solve this dilemma by defining our Q-values as a function of future Q-values. This relation is called the *Bellman equation*, and it states that the maximum future reward for taking action is the current reward plus the next step's max future reward from taking the next action a':

$$Q^*(s_t, a_t) = E\left[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a')\right]$$

This recursive definition allows us to relate between Q-values.

And since we can now relate between Q-values past and future, this equation conveniently defines an update rule. Namely, we can update past Q-values to be based on future Q-values. This is powerful because there exists a Q-value we know is correct: the Q-value of the very last action before the episode is over. For this last state, we know exactly that the next action led to the next reward, so we can perfectly set the Q-values for that state. We can use the update rule, then, to propagate that Q-value to the previous time step:

$$\widehat{Q_j} \rightarrow \widehat{Q_{j+1}} \rightarrow \widehat{Q_{j+2}} \rightarrow \ldots \rightarrow Q^*$$

This updating of the Q-value is known as *value iteration*.

Our first Q-value starts out completely wrong, but this is perfectly acceptable. With each iteration, we can update our Q-value via the correct one from the future. After one iteration, the last Q-value is accurate, since it is just the reward from the last state and action before episode termination. Then we perform our Q-value update, which sets the second-to-last Q-value. In our next iteration, we can guarantee that the last two Q-values are correct, and so on and so forth. Through value iteration, we will be guaranteed convergence on the ultimate optimal Q-value.

Issues with Value Iteration

Value iteration produces a mapping between state and action pairs with corresponding Q-values, and we are constructing a table of these mappings, or a *Q-table*. Lets briefly talk about the size of this Q-table. Value iteration is an exhaustive process that requires a full traversal of the entire space of state, action pairs. In a game like Breakout, with 100 bricks that can be either present or not, with 50 positions for the paddle to be in, and 250 positions for the ball to be in, and 3 actions, we have already constructed a space that is far, far larger than the sum of all computational capacity of humanity. Furthermore, in stochastic environments, the space of our Q-table would be even larger, and possibly infinite. With such a large space, it will be intractable for us to find all of the Q-values for every state, action pair. Clearly this approach is not going to work. How else are we going to do Q-learning?

Approximating the Q-Function

The size of our Q-table makes the naive approach intractable for any nontoy problem. However, what if we relax our requirement for an optimal Q-function? If instead, we learn approximations of the Q-function, we can use a model to estimate our Q-function. Instead of having to experience every state, action pair to update our Q-table, we can learn a function that approximates this table, and even generalizes outside of its own experience. This means we won't have to perform an exhaustive search through all possible Q-values to learn a Q-function.

Deep Q-Network (DQN)

This was the main motivation behind DeepMind's work on Deep Q-Network (DQN). DQN uses a deep neural network that takes an image (the state) in to estimate the Q-value for all possible actions.

Training DQN

We would like to train our network to approximate the Q-function. We express this Q-function approximation as a function of our model's parameters, like this:

$$\widehat{Q_\theta}(s, a \mid \theta) \sim Q^\star(s, a)$$

Remember, Q-learning is a value-learning algorithm. We are not learning a policy directly, but rather we are learning the values of each state, action pair, regardless if they are good or not. We have expressed our model's Q-function approximation as Qtheta, and we would like this to be close to the future expected reward. Using the Bellman Equation from earlier, we can express this future expected reward as:

$$R_t^\star = \left(r_t + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a' \mid \theta)\right)$$

Our objective is to minimize the difference between our Q's approximation, and the next Q value:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \widehat{Q}(s_t, a_t \mid \theta) - R_t^\star$$

Expanding this expression gives us our full objective:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \widehat{Q}(s_t, a_t \mid \theta) - \left(r_t + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a' \mid \theta)\right)$$

This objective is fully differentiable as a function of our model parameters, and we can find gradients to use in stochastic gradient descent to minimize this loss.

Learning Stability

One issue you may have noticed is that we are defining our loss function based on the difference of our model's predicted Q-value of this step and the predicted Q-value of the next step. In this way our loss is doubly dependent on our model parameters. With each parameter update, the Q-values are constantly shifting, and we are using shifting Q-values to do further updates. This high correlation of updates can lead to feedback loops and instability in our learning where our parameters may oscillate and make the loss diverge.

We can employ a couple of simple engineering hacks to remedy this correlation problem; namely, target Q-network and experience replay.

Target Q-Network

Instead of updating a single network frequently with respect to itself, we can reduce this codependence by introducing a second network, called the *target network*. Our loss function features to instances of the Q-function, $\widehat{Q}(s_t, a_t | \theta)$ and $\widehat{Q}(s_{t+1}, a' | \theta)$. We are going to have the first Q be represented as our prediction network, and our second Q will be produced by the target Q-network. The target Q-network is a copy of our prediction network that lags in its parameter updates. We only update the target Q-network to equal the prediction network every few batches. This provides much needed stability to our Q-values, and we can now properly learn a good Q-function.

Experience Replay

There is yet another source of irksome instability to our learning: the high correlations of recent experiences. If we train our DQN with batches drawn from recent experience, these action, state pairs are all going to be related to one another. This is harmful because we want our batch gradients to be representative of the entire gradient, and if our data is not representative of the data distribution, our batch gradient will not be an accurate estimate of the true gradient.

So we have to break up this correlation of data in our batches. We can do this using something called *experience replay*. In experience replay, we store all of the agent's experiences as a table, and to construct a batch, we randomly sample from these experience. We store these experiences in a table as (s_i, a_i, r_i, s_{i+1}) tuples. From these four values, we can compute our loss function, and thus our gradient to optimize our network.

This experience replay table is more of a queue than a table. The experiences an agent sees early in training may not be representative of the experiences a trained agent finds itself in later, so it is useful to remove very old experiences from our table.

From Q-Function to Policy

Q-learning is a value learning paradigm, not a policy learning algorithm. This means we are not directly learning a policy for acting in our environment. But can't we construct a policy from what our Q-function tells us? If we have learned a good Q-function approximation, this means we know the value of every action for every state. We could then trivially construct an optimal policy in the following way: look at our Q-function for all actions in our current state, choose the action with the max Q-value, enter a new state, and repeat. If our Q-function is optimal, our policy derived from it will be optimal. With this in mind, we can express the optimal policy as follows:

$$\pi(s;\theta) = \arg\max_{a'} \widehat{Q^*}(s,a';\theta)$$

We can also use the sampling techniques we discussed earlier to make a stochastic policy that sometime deviates from the Q-function recommendations to vary the amount of exploration our agent does.

DQN and the Markov Assumption

DQN is still a Markov decision process that relies on the *Markov assumption*, which assumes that the next state s_i+1 depends only on the current state s_i and action a_i, and not on any previous states or actions. This assumption doesn't hold true for many environments where the game's state cannot be summed up in a single frame. For example, in Pong, the ball's velocity (an important factor in successful gameplay) is not captured in any single game frame. The Markov assumption makes modeling decision processes much simpler and reliable, but often at a loss of modeling power.

DQN's Solution to the Markov Assumption

DQN solves this problem by utilizing *state history*. Instead of processing one game frame as the game's state, DQN considers the past four game frames as the game's current state. This allows DQN to utilize time-dependent information. This is a bit of an engineering hack, and we will discuss better ways of dealing with sequences of states at the end of this chapter.

Playing Breakout wth DQN

Lets pull all of what we learned together and actually go about implementing DQN to play Breakout. First we start out by defining our DQNAgent:

```
# DQNAgent

class DQNAgent(object):

    def __init__(self, session, num_actions,
                 learning_rate=1e-3, history_length=4,
                 screen_height=84, screen_width=84,
                 gamma=0.98):
        self.session = session
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.history_length = history_length
        self.screen_height = screen_height
        self.screen_width = screen_width
        self.gamma = gamma

        self.build_prediction_network()
        self.build_target_network()
        self.build_training()
```

```
    def build_prediction_network(self):
        with tf.variable_scope('pred_network'):
            self.s_t = tf.placeholder('float32', shape=[
                                      None,
                                      self.history_length,
                                      self.screen_height,
                                      self.screen_width],
                                      name='state')
            self.conv_0 = slim.conv2d(self.s_t, 32, 8, 4,
              scope='conv_0')
            self.conv_1 = slim.conv2d(self.conv_0, 64, 4, 2,
              scope='conv_1')
            self.conv_2 = slim.conv2d(self.conv_1, 64, 3, 1,
              scope='conv_2')

            shape = self.conv_2.get_shape().as_list()

            self.flattened = tf.reshape(
                self.conv_2, [-1, shape[1]*shape[2]*shape[3]])
            self.fc_0 = slim.fully_connected(self.flattened,
               512, scope='fc_0')
            self.q_t = slim.fully_connected(
              self.fc_0, self.num_actions, activation_fn=None,
               scope='q_values')

            self.q_action = tf.argmax(self.q_t, dimension=1)

    def build_target_network(self):
        with tf.variable_scope('target_network'):
            self.target_s_t = tf.placeholder('float32',
              shape=[None, self.history_length,
                self.screen_height, self.screen_width],
                  name='state')
            self.target_conv_0 = slim.conv2d(
                self.target_s_t, 32, 8, 4, scope='conv_0')
            self.target_conv_1 = slim.conv2d(
                self.target_conv_0, 64, 4, 2, scope='conv_1')
            self.target_conv_2 = slim.conv2d(
                self.target_conv_1, 64, 3, 1, scope='conv_2')

            shape = self.conv_2.get_shape().as_list()

            self.target_flattened = tf.reshape(
                self.target_conv_2, [-1,
                  shape[1]*shape[2]*shape[3]])
            self.target_fc_0 = slim.fully_connected(
                self.target_flattened, 512, scope='fc_0')
            self.target_q = slim.fully_connected(
                self.target_fc_0, self.num_actions,
                  activation_fn=None, scope='q_values')
```

```
    def update_target_q_weights(self):
        pred_vars = tf.get_collection(
            tf.GraphKeys.GLOBAL_VARIABLES, scope=
              'pred_network')
        target_vars = tf.get_collection(
            tf.GraphKeys.GLOBAL_VARIABLES, scope=
              'target_network')
        for target_var, pred_var in zip(target_vars, pred_vars):
            weight_input = tf.placeholder('float32',
              name='weight')
            target_var.assign(weight_input).eval(
                {weight_input: pred_var.eval()})

    def build_training(self):
        self.target_q_t = tf.placeholder('float32', [None],
          name='target_q_t')
        self.action = tf.placeholder('int64', [None],
          name='action')

        action_one_hot = tf.one_hot(
            self.action, self.num_actions, 1.0, 0.0,
              name='action_one_hot')
        q_of_action = tf.reduce_sum(
            self.q_t * action_one_hot, reduction_indices=1,
              name='q_of_action')

        self.delta = tf.square((self.target_q_t - q_of_action))
        self.loss = tf.reduce_mean(self.delta, name='loss')

        self.optimizer = tf.train.AdamOptimizer(
            learning_rate=self.learning_rate)
        self.train_step = self.optimizer.minimize(self.loss)

    def sample_action_from_distribution(self,
      action_distribution, epsilon_percentage):
        # Choose an action based on the action probability
        # distribution
        action = epsilon_greedy_action_annealed(
            action_distribution, epsilon_percentage)
        return action

    def predict_action(self, state, epsilon_percentage):
        action_distribution = self.session.run(
            self.q_t, feed_dict={self.s_t: [state]})[0]
        action = self.sample_action_from_distribution(
            action_distribution, epsilon_percentage)
        return action

    def process_state_into_stacked_frames(self, frame,
      past_frames, past_state=None):
        full_state = np.zeros(
            (self.history_length, self.screen_width,
```

```
            self.screen_height))

        if past_state is not None:
            for i in range(len(past_state)-1):
                full_state[i, :, :] = past_state[i+1,
                  :, :]
            full_state[-1, :, :] = imresize(to_grayscale(frame),
                                            (self.screen_width,
                                            self.screen_height))
                                              /255.0
        else:
            all_frames = past_frames + [frame]
            for i, frame_f in enumerate(all_frames):
                full_state[i, :, :] = imresize(
                    to_grayscale(frame_f), (self.screen_width,
                      self.screen_height))/255.0
            full_state = full_state.astype('float32')
        return full_state
```

There is a lot going on in this class, so let's break it down.

Building Our Architecture

We build our two Q-networks: the prediction network and the target Q-network. Notice how they have the same architecture definition, since they are the same network, with the target Q just having delayed parameter updates. Since we are learning to play Breakout from pure pixel input, our game state is an array of pixels. We pass this image through three convolution layers, and then two fully connected layers to produce our Q-values for each of our potential actions.

Stacking Frames

You may notice that our state input is actually of size `[None, self.history_length, self.screen_height, self.screen_width]`. Remember, in order to model and capture time-dependent state variables like speed, DQN uses not just one image, but a group of consecutive images, also known as a *history*. Each of these consecutive images is treated as a separate channel. We construct these stacked frames with the helper function `process_state_into_stacked_frames(self, frame, past_frames, past_state=None)`.

Setting Up Training Operations

Our loss function is derived from our objective expression from earlier in this chapter:

$$\min_{\theta} \sum_{e \in E} \sum_{t=0}^{T} \widehat{Q}(s_t, a_t | \theta) - \left(r_t + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a' | \theta)\right)$$

We want our prediction network to equal our target network, plus the return at the current time step. We can express this in pure TensorFlow code as the difference between the output of our prediction network and the output of our target network. We use this gradient to update and train our prediction network, using `AdamOptimizer`.

Updating Our Target Q-Network

To ensure a stable learning environment, we only update our target Q-network once every four batches. Our update rule for the target Q-network is pretty simple: we just set its weights equal to the prediction network. We do this in the function `update_target_q_network(self)`. We can use `tf.get_collection()` to grab the variables of the prediction and target network scopes. We can loop through these variables and run the `tf.assign()` operation to set the target Q-network's weights equal to those of the prediction network.

Implementing Experience Replay

We've discussed how experience replay can help de-correlate our gradient batch updates to improve our the quality of our Q-learning and subsequent derived policy. Let's walk though a simple implementation of experience replay. We expose a method `add_episode(self, episode)` which takes an entire episode (an `EpisodeHistory` object) and adds it to the ExperienceReplayTable. It then checks if the table is full and removes the oldest experiences from the table.

When it comes time to sample from this table, we can call `sample_batch(self, batch_size)` to randomly construct a batch from our table of experiences:

```
class ExperienceReplayTable(object):

    def __init__(self, table_size=5000):
        self.states = []
        self.actions = []
        self.rewards = []
        self.state_primes = []
        self.discounted_returns = []

        self.table_size = table_size

    def add_episode(self, episode):
        self.states += episode.states
        self.actions += episode.actions
        self.rewards += episode.rewards
        self.discounted_returns += episode.discounted_returns
        self.state_primes += episode.state_primes

        self.purge_old_experiences()
```

```
    def purge_old_experiences(self):
        if len(self.states) > self.table_size:
            self.states = self.states[-self.table_size:]
            self.actions = self.actions[-self.table_size:]
            self.rewards = self.rewards[-self.table_size:]
            self.discounted_returns = self.discounted_returns[
                -self.table_size:]
            self.state_primes = self.state_primes[
              -self.table_size:]

    def sample_batch(self, batch_size):
        s_t, action, reward, s_t_plus_1, terminal = [], [],
          [], [], []
        rands = np.arange(len(self.states))
        np.random.shuffle(rands)
        rands = rands[:batch_size]
        for r_i in rands:
            s_t.append(self.states[r_i])
            action.append(self.actions[r_i])
            reward.append(self.rewards[r_i])
            s_t_plus_1.append(self.state_primes[r_i])
            terminal.append(self.discounted_returns[r_i])
        return np.array(s_t), np.array(action),
            np.array(reward), np.array(s_t_plus_1),
            np.array(terminal)
```

DQN Main Loop

Let's put this all together in our main function, which will create an OpenAI Gym environment for Breakout, make an instance of our DQNAgent, and have our agent interact with and train to play Breakout successfully:

```
def main(argv):
    # Configure Settings
    run_index = 0
    learn_start = 100
    scale = 10
    total_episodes = 500*scale
    epsilon_stop = 250*scale
    train_frequency = 4
    target_frequency = 16
    batch_size = 32
    max_episode_length = 1000
    render_start = total_episodes - 10
    should_render = True

    env = gym.make('Breakout-v0')
    num_actions = env.action_space.n

    solved = False
    with tf.Session() as session:
        agent = DQNAgent(session=session,
```

```
  num_actions=num_actions)
session.run(tf.global_variables_initializer())

episode_rewards = []
batch_losses = []

replay_table = ExperienceReplayTable()
global_step_counter = 0
for i in tqdm.tqdm(range(total_episodes)):
    frame = env.reset()
    past_frames = [frame] * (agent.history_length-1)
    state = agent.process_state_into_stacked_frames(
        frame, past_frames, past_state=None)
    episode_reward = 0.0
    episode_history = EpisodeHistory()
    epsilon_percentage = float(min(i/float(
      epsilon_stop), 1.0))
    for j in range(max_episode_length):
        action = agent.predict_action(state,
          epsilon_percentage)
        if global_step_counter < learn_start:
            action = random_action(agent.num_actions)

        # print(action)
        frame_prime, reward, terminal, _ = env.step(
          action)
        state_prime =
          agent.process_state_into_stacked_frames(
            frame_prime, past_frames,
              past_state=state)

        past_frames.append(frame_prime)
        past_frames = past_frames[-4:]

        if (render_start > 0 and (i >
          render_start)
              and should_render) or (solved and
                should_render):
            env.render()
        episode_history.add_to_history(
            state, action, reward, state_prime)
        state = state_prime
        episode_reward += reward
        global_step_counter += 1
        if j == (max_episode_length - 1):
            terminal = True

        if terminal:
            episode_history.discounted_returns =
              discount_rewards(
                episode_history.rewards)
            replay_table.add_episode(episode_history)
```

```
            if global_step_counter > learn_start:
                if global_step_counter %
                  train_frequency == 0:
                    s_t, action, reward, s_t_plus_1,
                      terminal = \
                        replay_table.sample_batch(
                          batch_size)
                    q_t_plus_1 = agent.target_q.eval(
                        {agent.target_s_t:
                          s_t_plus_1})

                    terminal = np.array(terminal) + 0.
                    max_q_t_plus_1 = np.max(q_t_plus_1,
                      axis=1)
                    target_q_t = (1. - terminal) * \
                        agent.gamma * max_q_t_plus_1 +
                          reward

                    _, q_t, loss = agent.session.run(
                        [agent.train_step, agent.q_t,
                          agent.loss], {
                        agent.target_q_t: target_q_t,
                        agent.action: action,
                        agent.s_t: s_t
                    })

                if global_step_counter %
                  target_frequency == 0:
                    agent.update_target_q_weights()

            episode_rewards.append(episode_reward)
            break

    if i % 50 == 0:
        ave_reward = np.mean(episode_rewards[-100:])
        print(ave_reward)
        if ave_reward > 50.0:
            solved = False
        else:
            solved = False
```

DQNAgent Results on Breakout

We train our DQNAgent for 1,000 episodes to see the learning curve. To obtain superhuman results on Atari, typical training time runs up to several days. However, we can see a general upward trend in reward pretty quickly, as shown in Figure 9-7.

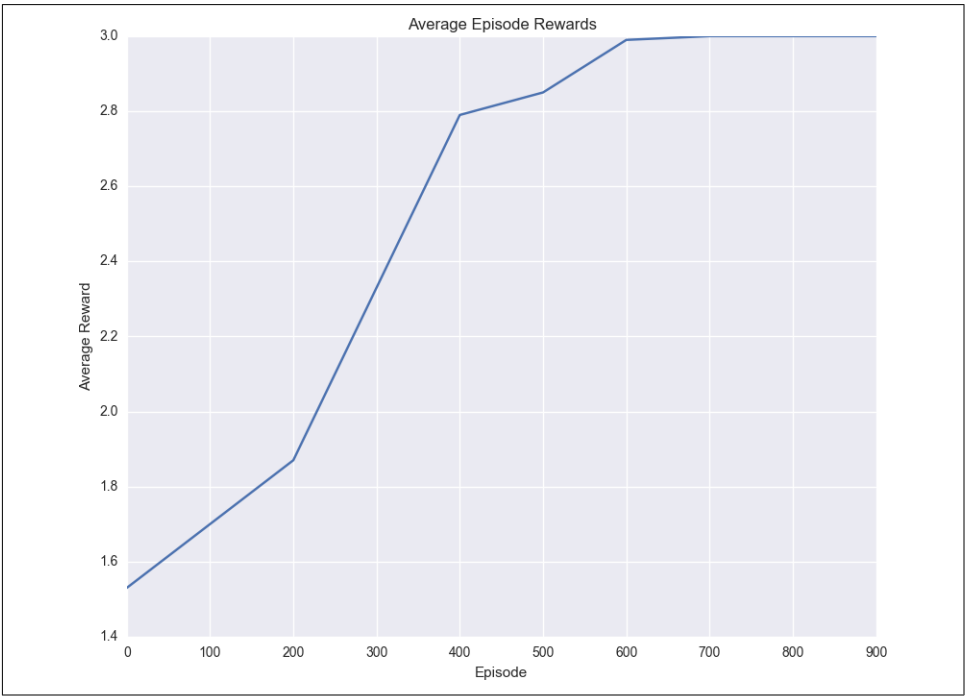


Figure 9-7. Our DQN agent gets increasingly better at Breakout during training as it learns a good value function and also acts less stochastically due to epsilon-greedy annealing

Improving and Moving Beyond DQN

DQN did a pretty good job back in 2013 in solving Atari tasks, but had some serious shortcomings. DQN's many weaknesses include that it takes very long to train, doesn't work well on certain types of games, and requires retraining for every new game. Much of the deep reinforcement learning research of the past few years has been in addressing these various weaknesses.

Deep Recurrent Q-Networks (DRQN)

Remember the Markov assumption? The one that states that the next state relies only on the previous state and the action taken by the agent? DQN's solution to the Markov assumption problem, stacking four consecutive frames as separate channels, sidesteps this issue and is a bit of an ad hoc engineering hack. Why four frames and not 10? This imposed frames history hyperparameter limits the model's generality. How do we deal with arbitrary sequences of related data? That's right: we can use what we learned back in Chapter 6 on recurrent neural networks to model sequences with *deep recurrent Q-networks* (DRQN).

DRQN uses a recurrent layer to transfer a latent knowledge of state from one time step to the next. In this way, the model itself can learn how many frames are informative to include in its state and can even learn to throw away noninformative ones or remember things from long ago.

DRQN has even been extended to include neural attention mechanism, as shown in Sorokin et al.'s 2015 paper "Deep Attention Recurrent Q-Network" (DAQRN).[5] Since DRQN is dealing with sequences of data, it can attend to certain parts of the sequence. This ability to attend to certain parts of the image both improves performance and provides model interpretability by producing a rationale for the action taken.

DRQN has shown to be better than DQN at playing first-person shooter (FPS) games like DOOM,[6] as well as improving performance on certain Atari games with long time-dependencies, like Seaquest.[7]

Asynchronous Advantage Actor-Critic Agent (A3C)

Asynchronous advantage actor-critic (A3C) is a new approach to deep reinforcement learning introduced in the 2016 DeepMind paper "Asynchronous Methods for Deep Reinforcement Learning."[8] Let's discuss what it is and why it improves upon DQN.

A3C is *asynchronous*, which means we can parallelize our agent across many threads, which means orders of magnitude faster training by speeding up our environment simulation. A3C runs many environments at once to gather experiences. Beyond the speed increase, this approach presents another significant advantage in that it further decorrelates the experiences in our batches, because the batch is being filled with the experiences of numerous agents in different scenarios simultaneously.

A3C uses an *actor-critic*[9] method. Actor-critic methods involve learning both a value function $V(s_t)$ (the critic) and also a policy $\pi(s_t)$, (the actor). Early in this chapter, we delineated two different approaches to reinforcement learning: value learning and policy learning. A3C combines the strengths of each, using the critic's value function to improve the actor's policy.

A3C uses an *advantage* function instead of a pure discounted future return. When doing policy learning, we want to penalize the agent when it chooses an action that

---

5 Sorokin, Ivan, et al. "Deep Attention Recurrent Q-Network." *arXiv preprint arXiv*:1512.01693 (2015).

6 *https://en.wikipedia.org/wiki/Doom_(1993_video_game)*

7 *https://en.wikipedia.org/wiki/Seaquest_(video_game)*

8 Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." *International Conference on Machine Learning*. 2016.

9 Konda, Vijay R., and John N. Tsitsiklis. "Actor-Critic Algorithms." *NIPS*. Vol. 13. 1999.

leads to a bad reward. A3C aims to achieve this same goal, but uses advantage instead of reward as its criterion. Advantage represents the difference between the model's prediction of the quality of the action taken versus the actual quality of the action taken. We can express advantage as:

$$A_t = Q^*(s_t, a_t) - V(s_t).$$

A3C has a value function, V(t), but it does not express a Q-function. Instead, A3C estimates the advantage by using the discounted future reward as an approximation for the Q-function:

$$A_t = R_t - V(s_t)$$

These three techniques proved key to A3C's takeover of most deep reinforcement learning benchmarks. A3C agents can learn to play Atari Breakout in less than 12 hours, whereas DQN agents may take 3 to 4 days.

UNsupervised REinforcement and Auxiliary Learning (UNREAL)

UNREAL is an improvement on A3C introduced in "Reinforcement learning with unsupervised auxiliary tasks" [10] by Jaderberg et al., who, you guessed it, are from DeepMind.

UNREAL addresses the problem of reward sparsity. Reinforcement learning is so difficult because our agent just receives rewards, and it is hard to determine exactly why rewards increase or decrease, which makes learning difficult. Additionally, in reinforcement learning, we must learn a good representation of the world as well as a good policy to achieve reward. Doing all of this with a weak learning signal like sparse rewards is quite a tall order.

UNREAL asks the question, what can we learn from the world without rewards, and aims to learn a useful world representation in an unsupervised matter. Specifically, UNREAL adds some additional unsupervised auxiliary tasks to its overall objective.

The first task involves the UNREAL agent learning about how its actions affect the environment. The agent is tasked with controlling pixel values on the screen by taking actions. To produce a set of pixel values in the next frame, the agent must take a specific action in this frame. In this way, the agent learns how its actions affect the world around it, enabling it to learn a representation of the world that takes into account its own actions.

The second task involves the UNREAL agent learning *reward prediction*. Given a sequence of states, the agent is tasked with predicting the value of the next reward

10 Jaderberg, Max, et al. "Reinforcement Learning with Unsupervised Auxiliary Tasks." *arXiv preprint arXiv*: 1611.05397 (2016).

received. The intuition behind this is that if an agent can predict the next reward, it probably has a pretty good model of the future state of the environment, which will be useful when constructing a policy.

As a result of these unsupervised auxiliary tasks, UNREAL is able to learn around 10 times faster than A3C on the Labyrynth game environment. UNREAL highlights the importance of learning good world representations and how unsupervised learning can aid in weak learning signal or low-resource learning problems like reinforcement learning.

Summary

In this chapter, we covered the fundamentals of reinforcement learning, including MDP's, maximum discounted future rewards, and explore versus exploit. We also covered various approaches to deep reinforcement learning, including policy gradients and Deep Q-Networks, and touched on some recent improvements on DQN and new developments in deep reinforcement learning.

Reinforcement learning is essential to building agents that can not only perceive and interpret the world, but also take action and interact with it. Deep reinforcement learning has made major advancements toward this goal, successfully producing agents capable of mastering Atari games, safely driving automobiles, trading stocks profitably, controlling robots, and more.

Index

D

E

M

N

O

P

Q

R

S

T

U

V

W