

O'REILLY®

Early Release
RAW & UNEDITED

Fundamentals of Deep Learning

DESIGNING NEXT-GENERATION
ARTIFICIAL INTELLIGENCE ALGORITHMS

Nikhil Buduma

Fundamentals of Deep Learning

*Designing Next Generation
Artificial Intelligence Algorithms*

This Preview Edition of *Fundamentals of Deep Learning*, Chapters 1–6, is a work in progress. The final book is expected to release on oreilly.com and through other retailers in December, 2016.

Nikhil Buduma

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Fundamentals of Deep Learning

by Nikhil Buduma

Copyright © 2015 Nikhil Buduma. All rights reserved.

Printed in the United States of America.

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

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

Editors: Mike Loukides and Shannon Cutt

Production Editor: FILL IN PRODUCTION EDITOR

TOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

November 2015: First Edition

Revision History for the First Edition

2015-06-12 First Early Release

2016-02-29: Second Early Release

2016-09-26: Third Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491925614> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Fundamentals of Deep Learning*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-92561-4

[FILL IN]

Table of Contents

1. The Neural Network.....	9
Building Intelligent Machines	9
The Limits of Traditional Computer Programs	10
The Mechanics of Machine Learning	11
The Neuron	15
Expressing Linear Perceptrons as Neurons	17
Feed-forward Neural Networks	18
Linear Neurons and their Limitations	21
Sigmoid, Tanh, and ReLU Neurons	21
Softmax Output Layers	23
Looking Forward	24
2. Training Feed-Forward Neural Networks.....	25
The Cafeteria Problem	25
Gradient Descent	27
The Delta Rule and Learning Rates	29
Gradient Descent with Sigmoidal Neurons	31
The Backpropagation Algorithm	33
Stochastic and Mini-Batch Gradient Descent	36
Test Sets, Validation Sets, and Overfitting	38
Preventing Overfitting in Deep Neural Networks	45
Summary	49
3. Implementing Neural Networks in TensorFlow	51
What is TensorFlow?	51
How Does TensorFlow Compare to Alternatives?	52
Installing TensorFlow	53
Creating and Manipulating TensorFlow Variables	55

TensorFlow Operations	57
Placeholder Tensors	58
Sessions in TensorFlow	59
Navigating Variable Scopes and Sharing Variables	60
Managing Models over the CPU and GPU	63
Specifying the Logistic Regression Model in TensorFlow	65
Logging and Training the Logistic Regression Model	68
Leveraging TensorBoard to Visualize Computation Graphs and Learning	70
Building a Multilayer Model for MNIST in TensorFlow	72
Summary	75
4. Beyond Gradient Descent.....	77
The Challenges with Gradient Descent	77
Local Minima in the Error Surfaces of Deep Networks	78
Model Identifiability	79
How Pesky are Spurious Local Minima in Deep Networks?	80
Flat Regions in the Error Surface	84
When the Gradient Points in the Wrong Direction	87
Momentum-Based Optimization	89
A Brief View of Second Order Methods	93
Learning Rate Adaptation	95
AdaGrad - Accumulating Historical Gradients	95
RMSProp - Exponentially Weighted Moving Average of Gradients	96
Adam - Combining Momentum and RMSProp	97
The Philosophy Behind Optimizer Selection	99
Summary	100
5. Convolutional Neural Networks.....	101
Neurons in Human Vision	101
The Shortcomings of Feature Selection	101
Vanilla Deep Neural Networks Don't Scale	105
Filters and Feature Maps	107
Full Description of the Convolutional Layer	113
Max Pooling	117
Full Architectural Description of Convolution Networks	118
Closing the Loop on MNIST with Convolutional Networks	120
Image Preprocessing Pipelines Enable More Robust Models	122
Accelerating Training with Batch Normalization	123
Building a Convolutional Network for CIFAR-10	126
Visualizing Learning in Convolutional Networks	129
Leveraging Convolutional Filters to Replicate Artistic Styles	133
Learning Convolutional Filters for Other Problem Domains	135

Summary	136
6. Embedding and Representation Learning.....	137
Learning Lower Dimensional Representations	137
Principal Component Analysis	138
Motivating the Autoencoder Architecture	141
Implementing an Autoencoder in TensorFlow	142
Denoising to Force Robust Representations	156
Sparsity in Autoencoders	160
When Context is More Informative than the Input Vector	163
The Word2Vec Framework	166
Implementing the Skip-Gram Architecture	169
Summary	175

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 days 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.

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

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, 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 just the trick. But let's say we want to do something slightly more interesting, like write a program to automatically read someone's handwriting.

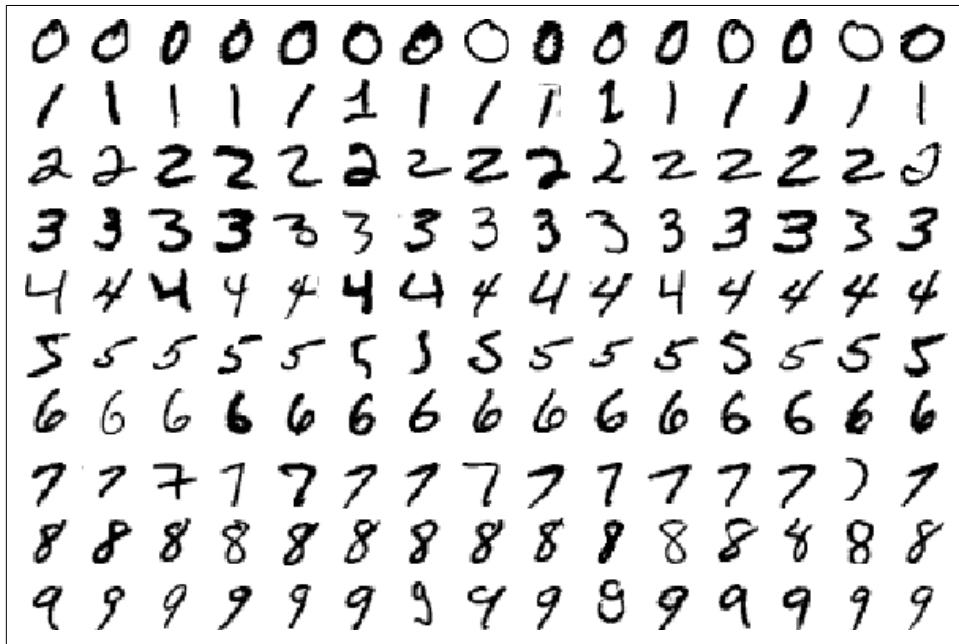


Figure 1-1. Image from MNIST handwritten digit dataset

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 a 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 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 difficult to distinguish from a six algorithmically

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.

There many other classes of problems that 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 example, 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're 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 billions of 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 the a massive list 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 greyscale image, the vector's components would be pixel intensities at each position, as shown in **Figure 1-3**.

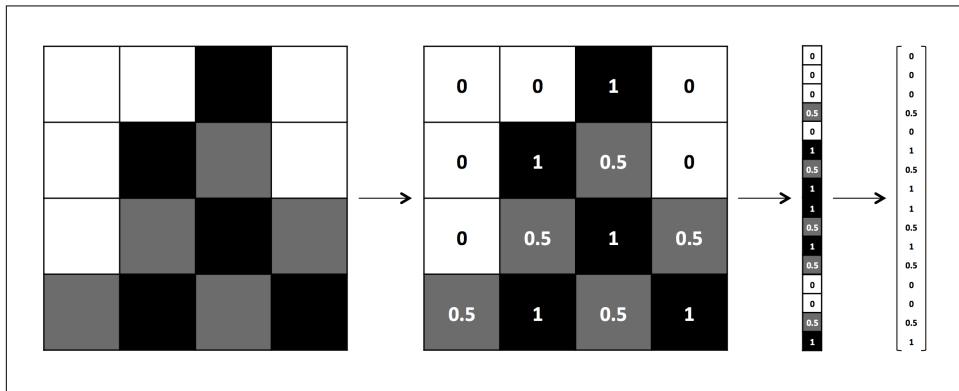


Figure 1-3. The process of vectorizing an image for a machine learning algorithm

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 in more detail in a future section.

To develop a more intuitive understanding for machine learning models, let's walk through a quick example. Let's say we wanted to figure out 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 $\mathbf{x} = [x_1 \ x_2]^T$, 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 average or below the class average. Our goal, then, might be to learn a model $h(\mathbf{x}, \theta)$ with parameter vector $\theta = [\theta_0 \ \theta_1 \ \theta_2]^T$ 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*. Let's assume our data is as shown in **Figure 1-4**.

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 in their performance is negligible. If this is not the case, we may want to collect more data to narrow our choice of θ .

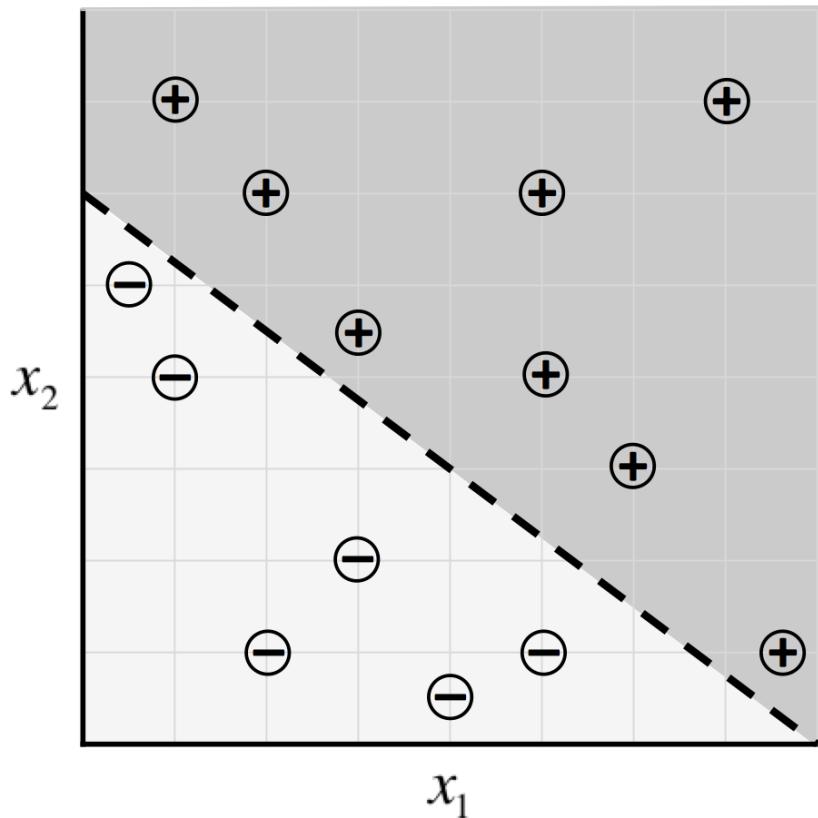


Figure 1-4. Sample data for our exam predictor algorithm and a potential classifier

This is pretty cool, but 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 the next chapter, when we describe the process of *gradient descent*. 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.

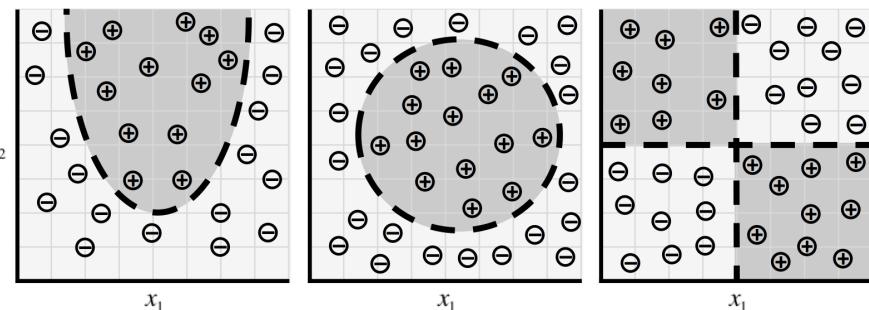


Figure 1-5. As our data takes on more complex forms, we need more complex models to describe them

But these situations are only the tip of the iceberg. As we move onto much more complex problems such as object recognition and text analysis, our data not only becomes extremely high dimensional, but the relationships we want to capture also become highly nonlinear. To accommodate this complexity, recent research in machine learning has attempted to build models that highly 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 a grain of rice, contains over 10,000 neurons, each of which forms an average of 6,000 connections with other neurons. 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 con-

nections, 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.

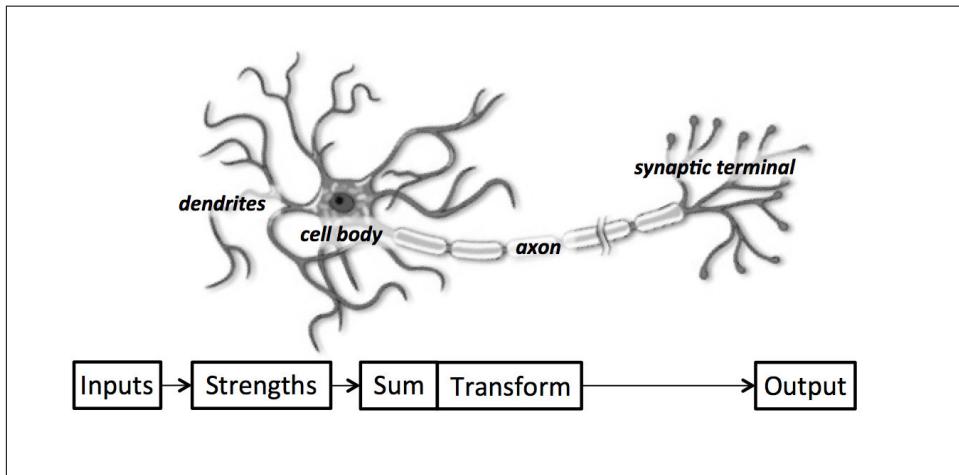


Figure 1-6. A functional description of a biological neuron's structure

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**. Just as in biological neurons, our artificial neuron takes in some number of inputs, x_1, x_2, \dots, x_n , each of which is multiplied by a specific weight, w_1, w_2, \dots, 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 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.

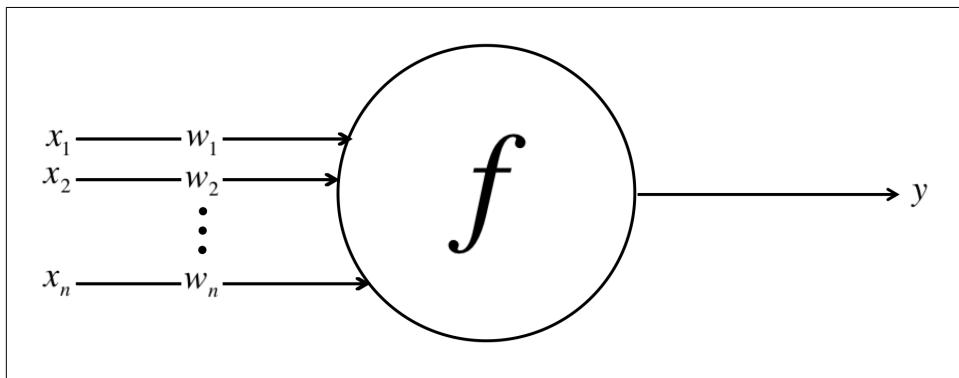


Figure 1-7. Schematic for a neuron in an artificial neural net

In **Example 1-1**, we show how a neuron might be implemented in Python. A few quick notes on implementation. Throughout this book, we'll be constantly using a couple of libraries to make our lives easier. One of these is *NumPy*, a fundamental library for scientific computing. Among other things, NumPy will allow us to quickly manipulate matrices and vectors with ease. In **Example 1-1**, NumPy enables us to painlessly take the dot product of two vectors (`inputs` and `self.weights`). Another library that we will use further down the road is *Theano*. Theano integrates closely with NumPy and allows us to define, optimize, and evaluate mathematical expressions. These two libraries will serve as a foundation for tools we explore in future chapters, so it's worth taking some time to gain some familiarity with them.

Example 1-1. Neuron Implementation

```
import numpy as np

#####
# Assume inputs and weights are 1-dimensional numpy #
# arrays and bias is a number                      #
#####

class Neuron:
    def __init__(self, weights, bias, function):
        self.weights = weights
        self.bias = bias
        self.function = function

    def forward(self, inputs):
        logit = np.dot(inputs, self.weights) + self.bias
        output = self.function(logit)
        return output
```

Expressing Linear Perceptrons as Neurons

In the previous section we talked about how 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:

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

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 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.

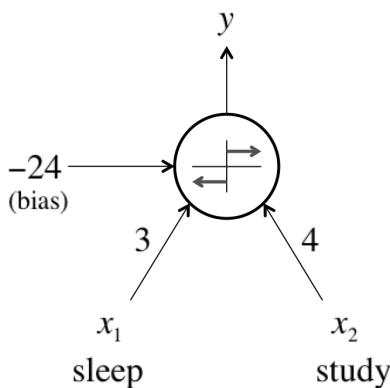


Figure 1-8. Expressing our exam performance perceptron as a neuron

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 hand-written 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 of six layers. Information flows from one layer to another until sensory input is converted into

conceptual understanding. For example, the bottom-most layer of the visual cortex receives raw visual data from the eyes. This information is processed by each layer and passed onto the next until, in the sixth layer, we conclude whether we are looking at a cat, or a soda can, or an airplane.

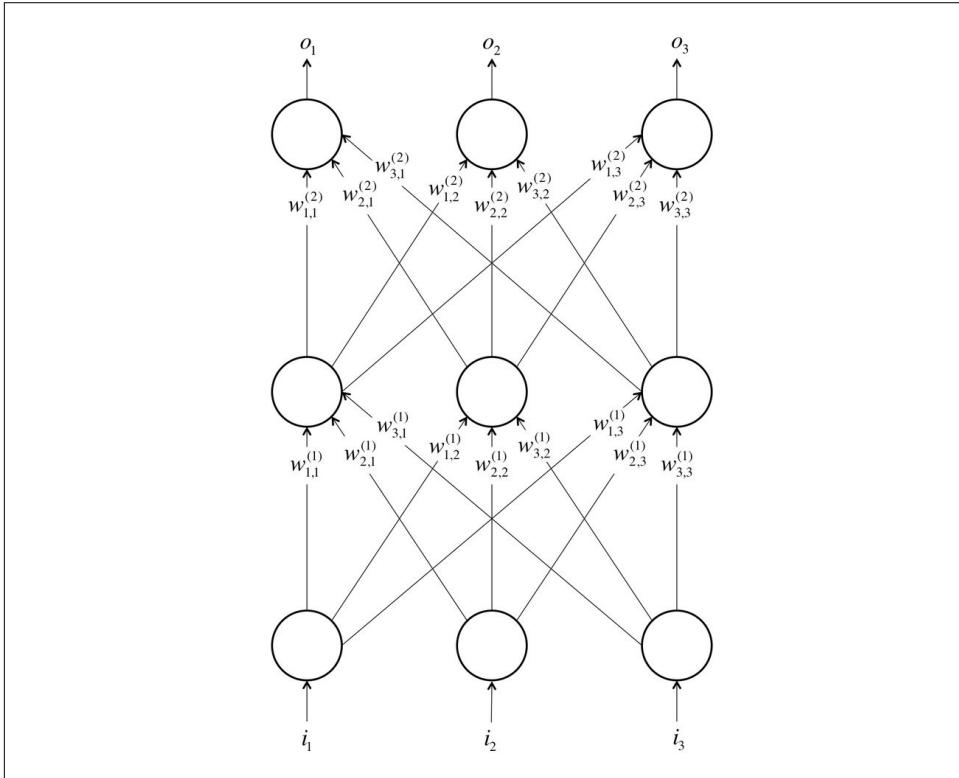


Figure 1-9. A simple example of a feed-forward neural network with 3 layers (input, one hidden, and output) and 3 neurons per layer

Borrowing 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. The 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 the next chapter. 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 above, 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. Often times, 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 often 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 2 neurons which 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.

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 cafeteria 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.

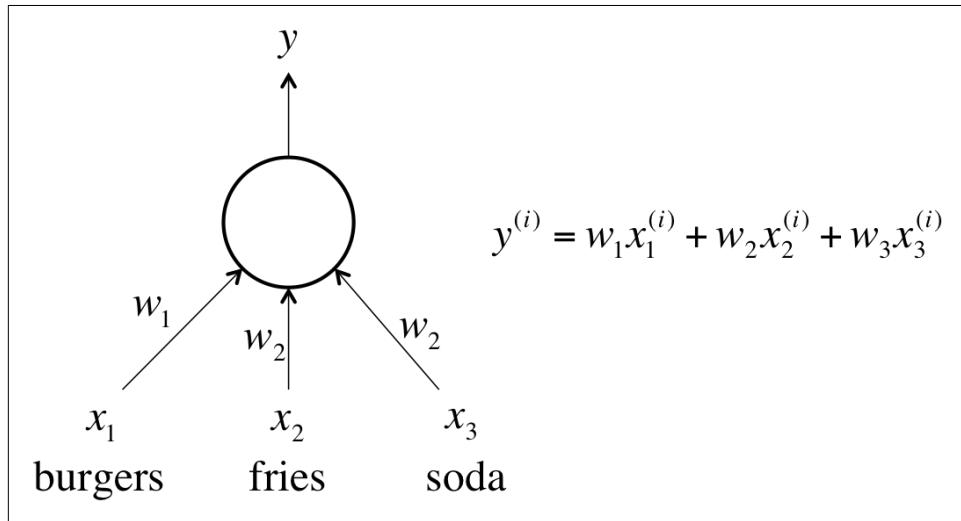


Figure 1-10. An example of a linear neuron

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:

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

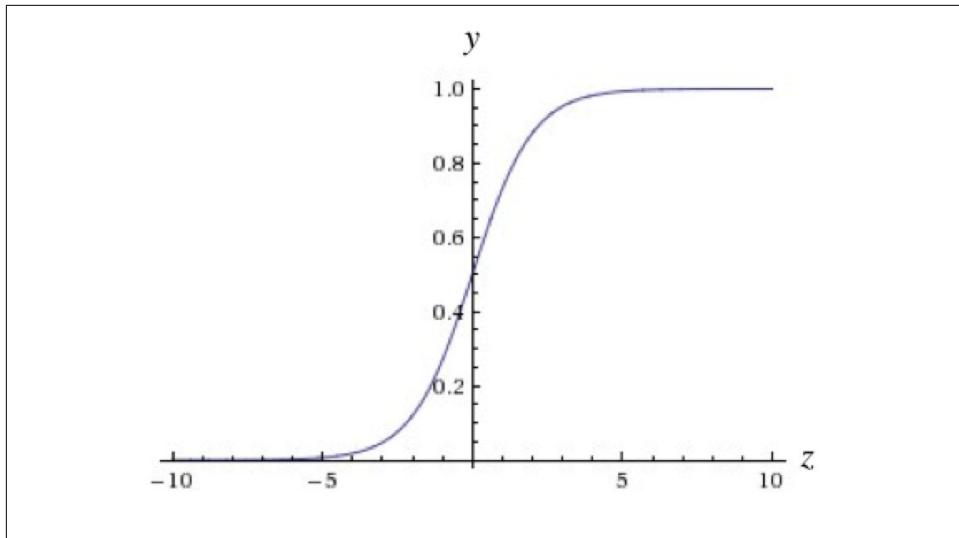


Figure 1-11. The output of a sigmoid neuron as z varies

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**.

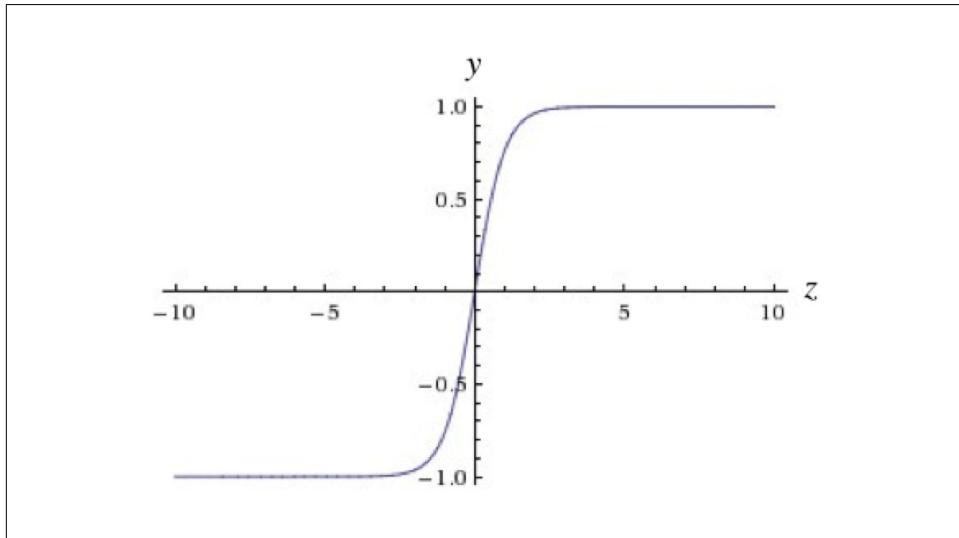


Figure 1-12. The output of a tanh neuron as z varies

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.

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**.

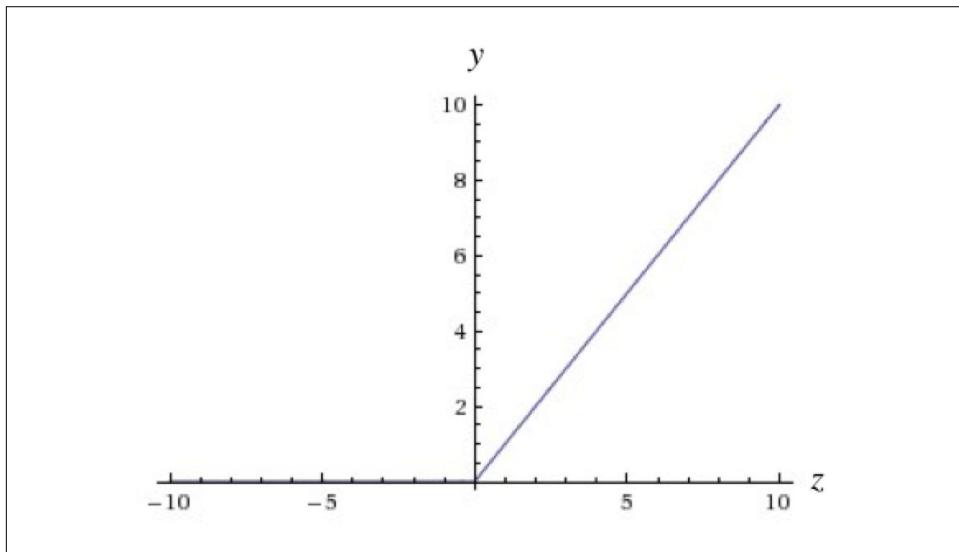


Figure 1-13. The output of a ReLU neuron as z varies

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

Softmax Output Layers

Often times, 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 data set. 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 $\sum_{i=0}^9 p_i = 1$:

$$[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_9]$$

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 of 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:

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

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.

Training Feed-Forward Neural Networks

The Cafeteria 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*. 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.

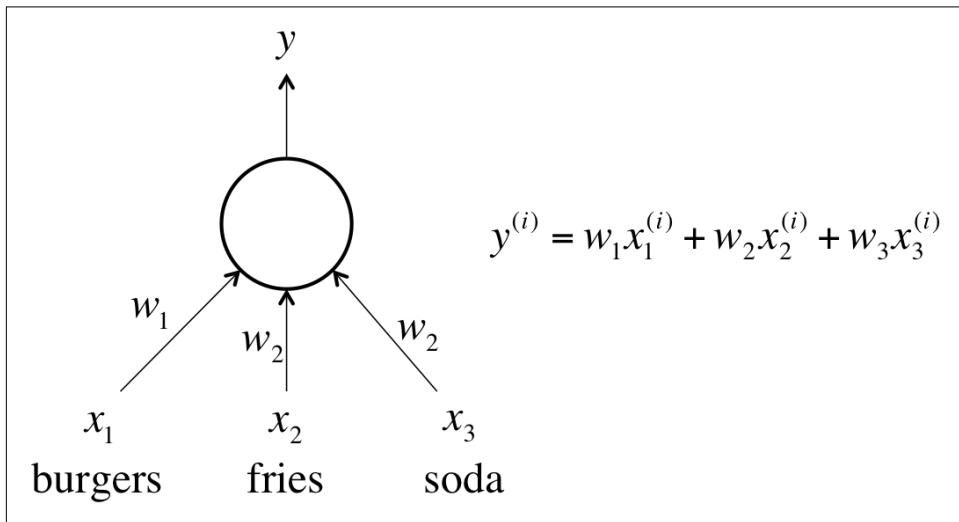


Figure 2-1. This is the neuron we want to train for the Dining Hall Problem

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 meal from the dining hall 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 smart 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, choosing smart training cases 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 this approach 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 bunch 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 :

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

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 example.

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 3-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 3-dimensional space, in particular, a quadratic bowl as shown in **Figure 2-2**.

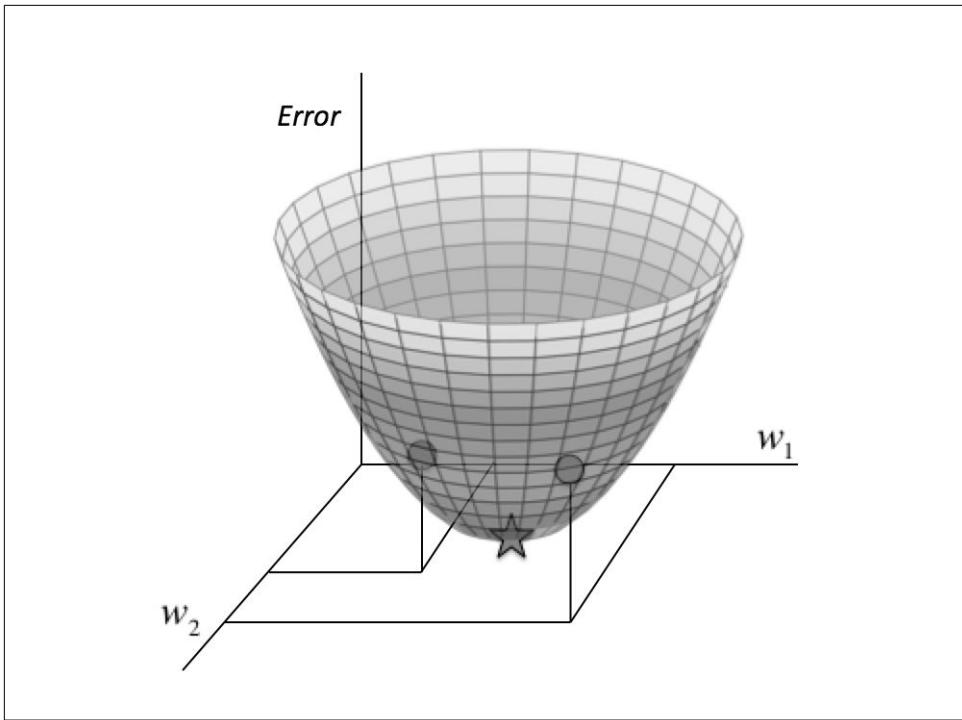


Figure 2-2. The quadratic error surface for a linear neuron

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 2-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*.

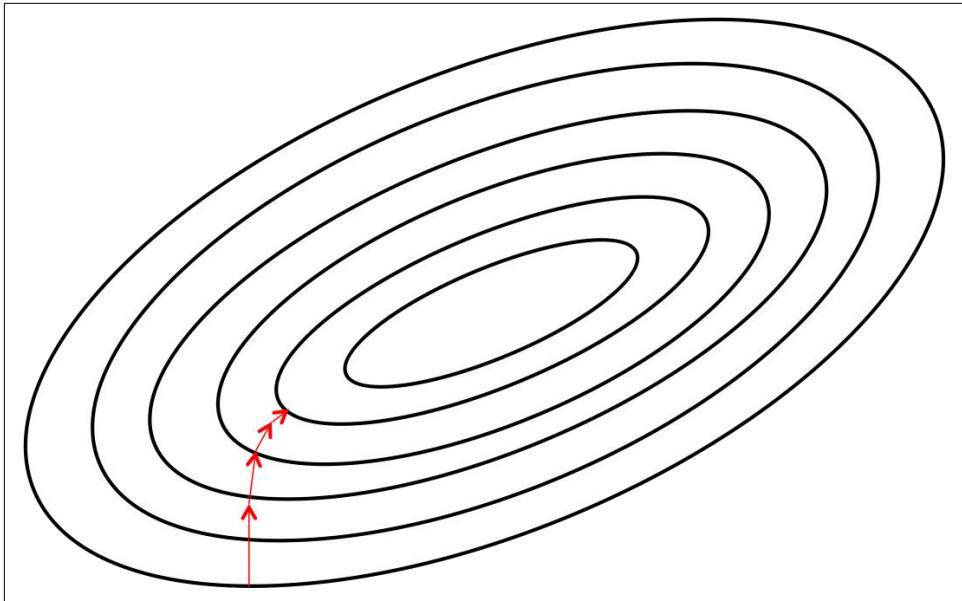


Figure 2-3. Visualizing the error surface as a set of contours

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 re-evaluate 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.

The Delta Rule and Learning Rates

Before we derive the exact algorithm for training our cafeteria neuron, we make 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. 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 the next chapter, we'll learn about various optimization techniques that utilize adaptive learning rates to automate the process of selecting learning rates.

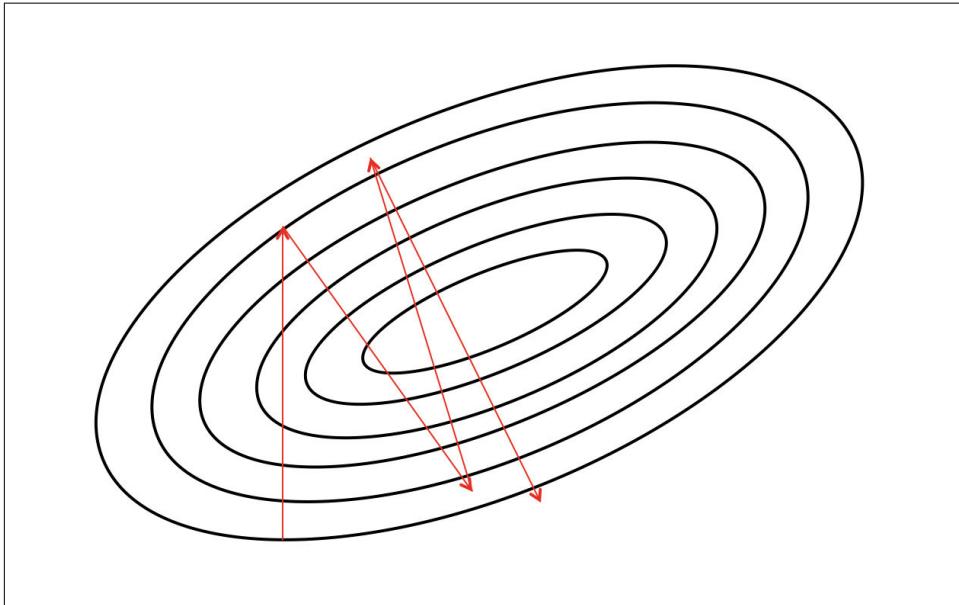


Figure 2-4. Convergence is difficult when our learning rate is too large

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:

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

$$\begin{aligned}
&= -\epsilon \frac{\partial}{\partial w_k} \left(\frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2 \right) \\
&= \sum_i \epsilon (t^{(i)} - y^{(i)}) \frac{\partial y_i}{\partial w_k} \\
&= \sum_i \epsilon x_k^{(i)} (t^{(i)} - y^{(i)})
\end{aligned}$$

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:

$$z = \sum_k w_k x_k$$

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

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.

$$\begin{aligned}
 \frac{dy}{dz} &= \frac{e^{-z}}{(1 + e^{-z})^2} \\
 &= \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} \\
 &= \frac{1}{1 + e^{-z}} \left(1 - \frac{1}{1 + e^{-z}}\right) \\
 &= y(1 - y)
 \end{aligned}$$

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:

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

Thus, the final rule for modifying the weights becomes:

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

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). 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 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 following diagram:

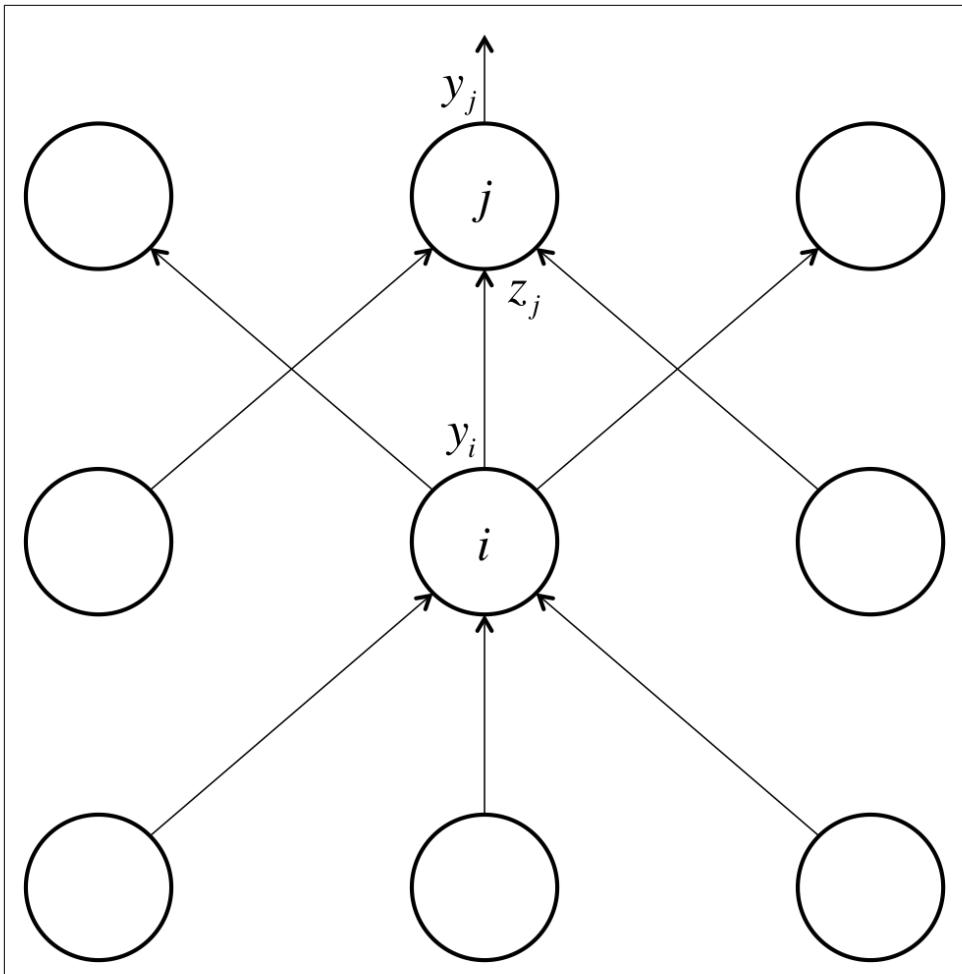


Figure 2-5. Reference diagram for the derivation of the backpropagation algorithm

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:

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

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} :

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

Furthermore, we observe the following:

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

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

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

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:

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

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

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

This completes our description of the backpropagation algorithm!

Stochastic and Mini-Batch Gradient Descent

In the algorithms we've described above, 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.

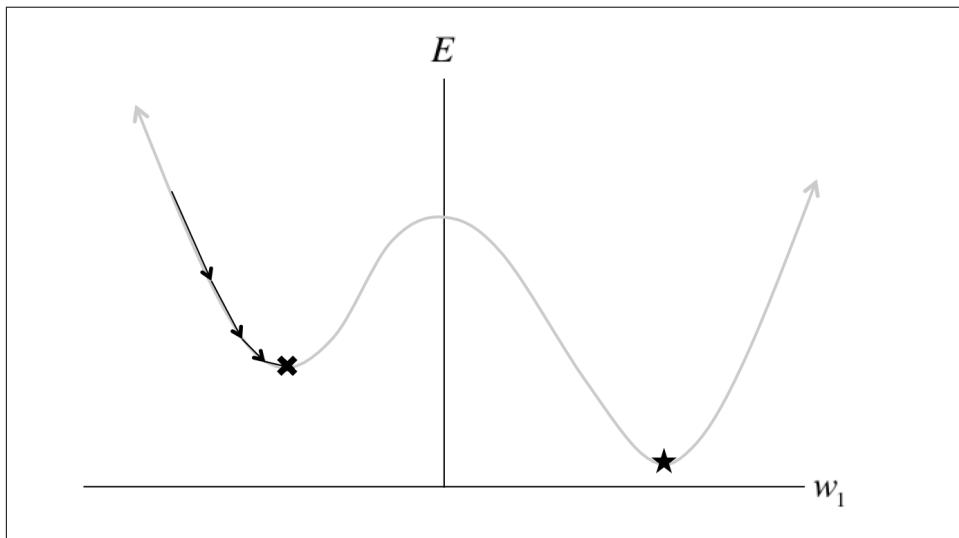


Figure 2-6. Batch gradient descent is sensitive to local minima

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 spurious local minimum, and if we get unlucky, we might get stuck in a non-optimal minima.

Another potential approach is *stochastic gradient descent*, 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 sur-

face is dynamic. As a result, descending on this stochastic surface significantly improves our ability to avoid local minima.

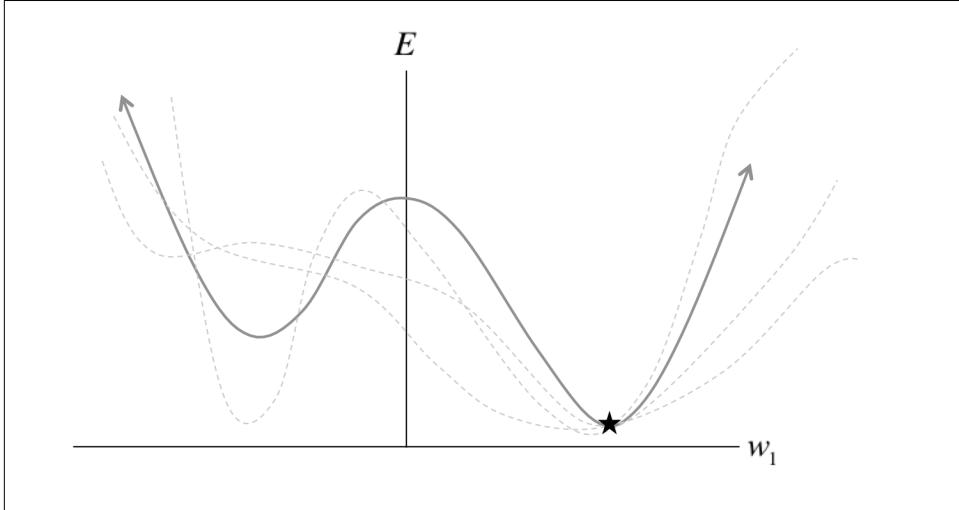


Figure 2-7. The stochastic error surface fluctuates with respect to the batch error surface, enabling local-minima avoidance

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 *mini-batch*, and in addition to the learning rate, mini-batch size is another hyperparameter. Mini-batches 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:

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

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 mini-batch.

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 by 28 pixels), feeds into two hidden layers with 30 neurons, and finally reaches a soft-max 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 take a look at the example data in **Figure 2-8**.

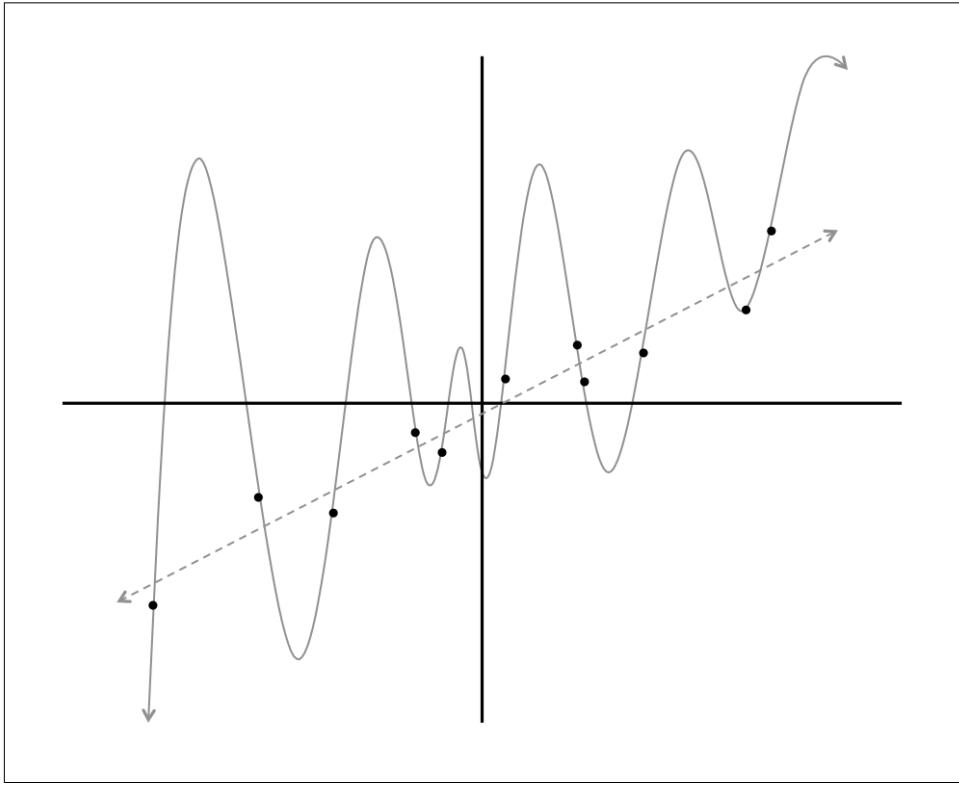


Figure 2-8. Two potential models that might describe our dataset - a linear model vs. a degree 12 polynomial

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**.

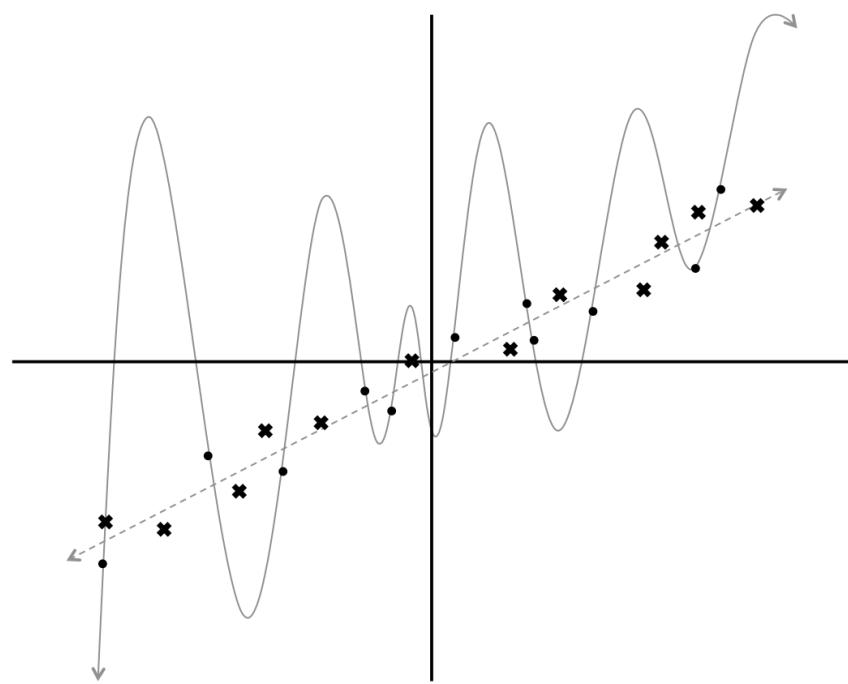


Figure 2-9. Evaluating our model on new data indicates that the linear fit is a much better model than the degree 12 polynomial

Now the verdict is clear, the linear model is not only subjectively better, but now also quantitatively performs better as well (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 dataset. 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 soft-max 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 the ConvnetJS library, are shown in **Figure 2-10**.

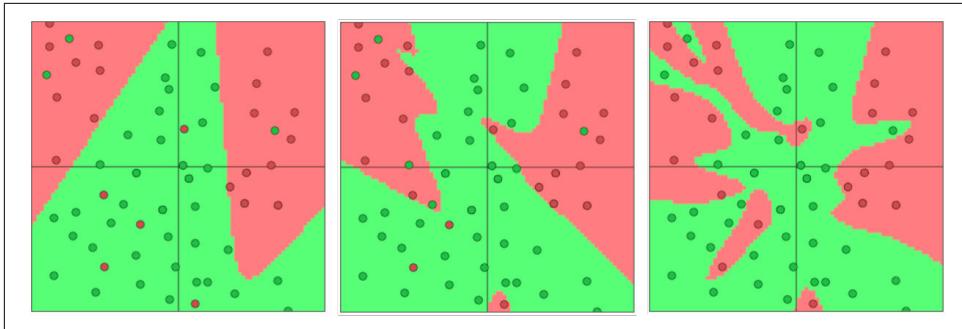


Figure 2-10. A visualization of neural networks with 3, 6, and 20 neurons (in that order) in their hidden layer.

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 1, 2, or 4 hidden layers of 3 neurons each.

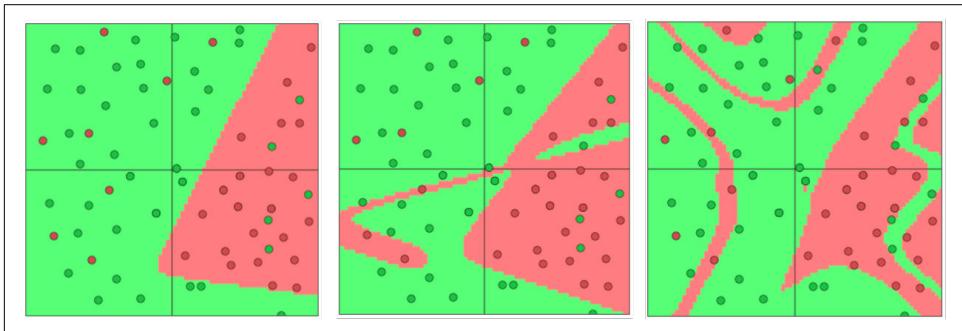


Figure 2-11. A visualization of neural networks with 1, 2, and 4 hidden layers (in that order) of 3 neurons each.

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.

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 polyno-

mial 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*.

Full Dataset:

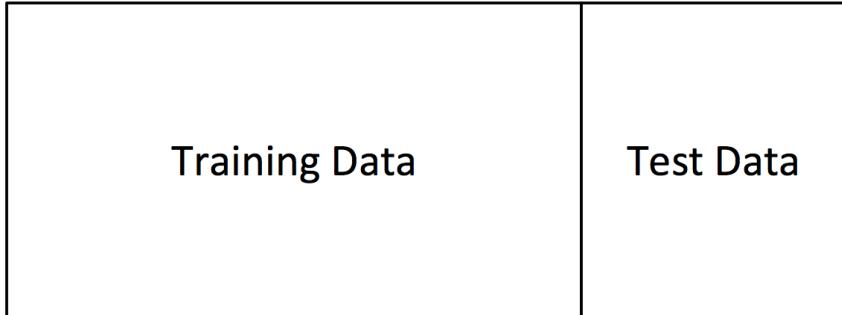


Figure 2-12. We often split our data into non-overlapping training and test sets in order to fairly evaluate our model

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. As a result, 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. As a result, 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.

Full Dataset:

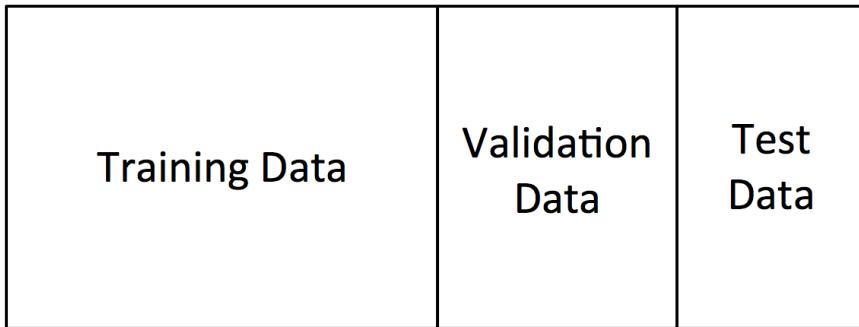


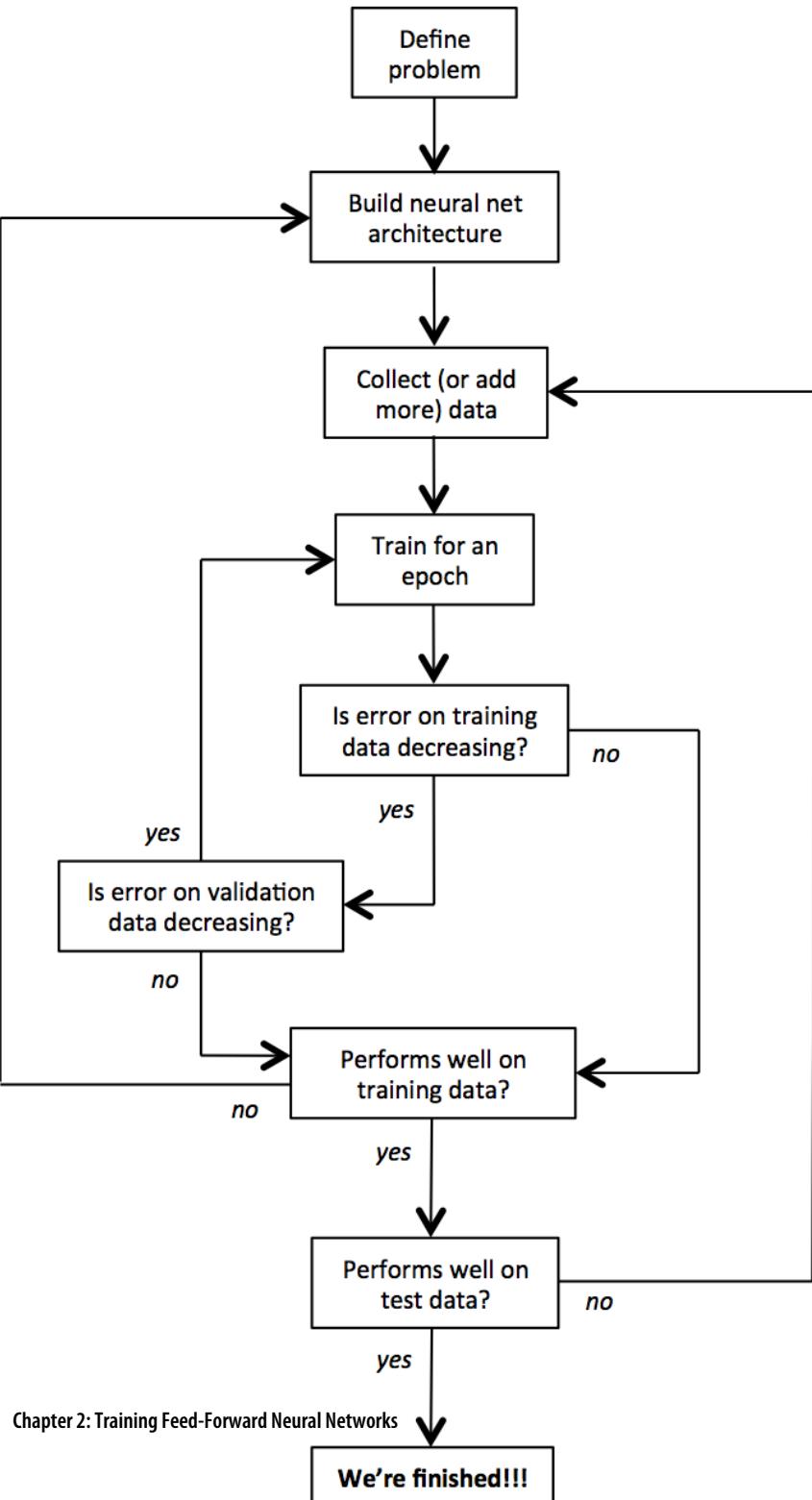
Figure 2-13. In deep learning we often include a validation set to prevent overfitting during the training process.

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 RGB 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).

After we build 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 model. 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.



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. 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, that means that 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 $\text{Error} + \lambda f(\theta)$, 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 is *L2 regularization*. 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, sing the L2 regularization ultimately means that every

weight is decayed linearly to zero. Expressed succinctly in NumPy, this is equivalent to the line: $W += -\lambda * W$. Because of this phenomenon, L2 regularization is also commonly referred to as *weight decay*.

We can visualize the effects of L2 regularization using ConvnetJs. Similar to above, we use a neural network with two inputs, a soft-max output of size two, 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**.

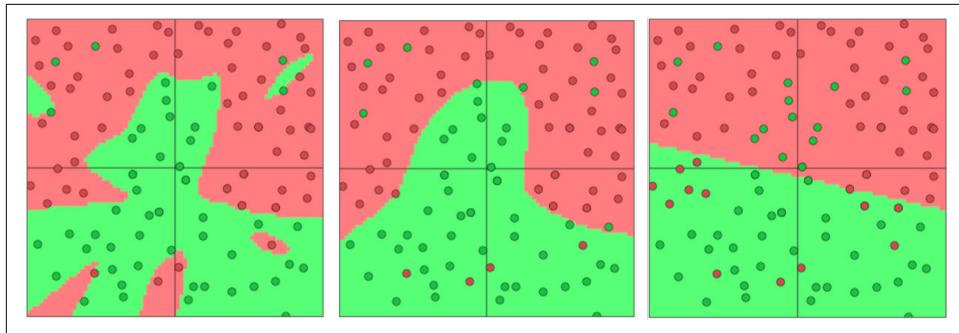


Figure 2-15. A visualization of neural networks trained with regularization strengths of 0.01, 0.1, and 1 (in that order).

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. 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, anytime a gradient descent step moved 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.

Dropout is a very different kind of method for preventing overfitting that can often be used in lieu of other techniques. 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**.

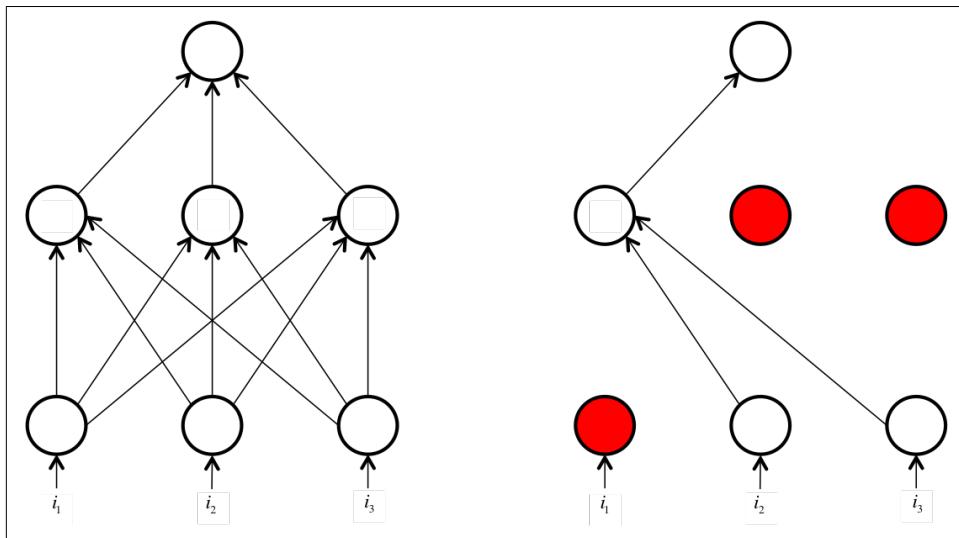


Figure 2-16. Dropout sets each neuron in the network as inactive with some random probability during each mini-batch of training.

Dropout is pretty intuitive to understand, but there are some important intricacies to consider. We illustrate these considerations through Python code. Let's assume we are working with a 3-layer ReLU neural network.

Example 2-1. Naïve Dropout Implementation

```
import numpy as np

# Let p = probability of keeping a hidden unit active
# A larger p means less dropout (p = 1 --> no dropout)
```

```

network.p = 0.5

def train_step(network, X):
    # forward pass for a 3-layer neural network

    Layer1 = np.maximum(0, np.dot(network.W1, X) + network.b1)
    # first dropout mask
    Dropout1 = (np.random.rand(*Layer1.shape) < network.p
    # first drop!
    Layer1 *= Dropout1

    Layer2 = np.maximum(0, np.dot(network.W2, Layer1) + network.b2)
    # second dropout mask
    Dropout2 = (np.random.rand(*Layer2.shape) < network.p
    # second drop!
    Layer2 *= Dropout2

    Output = np.dot(network.W3, Layer2) + network.b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(network, X):
    # NOTE: we scale the activations
    Layer1 = np.maximum(0, np.dot(network.W1, X) + network.b1) * network.p
    Layer2 = np.maximum(0, np.dot(network.W2, Layer1) + network.b2) * network.p
    Output = np.dot(network.W3, Layer2) + network.b3
    return Output

```

One of the things we'll realize is that during test-time (the `predict` function), we multiply the output of each layer by `network.p`. Why do we do this? Well, we'd like the outputs of neurons during test-time to be equivalent to their expected outputs at training 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 $\mathbb{E}[\text{output}] = px + (1 - p) \cdot 0 = px$.

The naïve implementation of dropout is undesirable 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 time. This has the additional appealing property that the `predict` code can remain the same whether or not dropout is used. In other words, only the `train_step` code would have to be modified.

Example 2-2. Inverted Dropout Implementation

```
import numpy as np

# Let network.p = probability of keeping a hidden unit active
# A larger network.p means less dropout (network.p == 1 --> no dropout)

network.p = 0.5

def train_step(network, X):
    # forward pass for a 3-layer neural network

    Layer1 = np.maximum(0, np.dot(network.W1, X) + network.b1)
    # first dropout mask, note that we divide by p
    Dropout1 = ((np.random.rand(*Layer1.shape) < network.p) / network.p
    # first drop!
    Layer1 *= Dropout1

    Layer2 = np.maximum(0, np.dot(network.W2, Layer1) + network.b2)
    # second dropout mask, note that we divide by p
    Dropout2 = ((np.random.rand(*Layer2.shape) < network.p) / network.p
    # second drop!
    Layer2 *= Dropout2

    Output = np.dot(network.W3, Layer2) + network.b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(network, X):
    Layer1 = np.maximum(0, np.dot(network.W1, X) + network.b1)
    Layer2 = np.maximum(0, np.dot(network.W2, Layer1) + network.b2)
    Output = np.dot(network.W3, Layer2) + network.b3
    return Output
```

Summary

In this chapter, we've learned all of the basics involved in training feedforward 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 Theano 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.

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 that you are also equipped with the skillsets 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 code.

The primary software tool that we will use throughout this text is called TensorFlow. 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 they are 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 are represented as tensors, which are multidimensional arrays. 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 it is 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 matrix multiplies) 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 multi-layer feedforward 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. 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. 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, to easily build models that span multiple GPUs on a single machine, and to 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 installed on our computers, we can use the following commands to install TensorFlow. Keep in mind the instructions are different if we'd like to use a GPU-enabled version of TensorFlow (which we strongly recommend). Unfortunately GPU support isn't present on OS X, but most Macs these days do not come with CUDA-enabled GPU's to begin with.

```
# Ubuntu/Linux 64-bit, CPU only:  
$ sudo pip install --upgrade https://storage.googleapis.com\  
 > /tensorflow/linux/cpu/tensorflow-0.5.0-cp27-none-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU enabled:  
$ sudo pip install --upgrade https://storage.googleapis.com\  
 > /tensorflow/linux/gpu/tensorflow-0.5.0-cp27-none-linux_x86_64.whl  
  
# Mac OS X, CPU only:  
$ sudo easy_install --upgrade six  
$ sudo pip install --upgrade https://storage.googleapis.com\  
 > /tensorflow/mac/tensorflow-0.5.0-py2-none-any.whl
```

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 7.0 (<https://developer.nvidia.com/cuda-toolkit-70>) and the CUDNN Toolkit 6.5 (<https://developer.nvidia.com/rdp/cudnn-archive>). 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-6.5-linux-x64-v2.tgz  
$ sudo cp cudnn-6.5-linux-x64-v2/cudnn.h /usr/local/cuda/include  
$ sudo cp cudnn-6.5-linux-x64-v2/libcudnn* /usr/local/cuda/lib64
```

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
```

If you'd like to install TensorFlow in a different way, several alternatives are listed here: https://www.tensorflow.org/versions/0.6.0/get_started/os_setup.html

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 are immediately wiped clean afterwards, variables survive across multiple executions of a graph. As a result, TensorFlow variables have the following three properties:

1. Variables must be explicitly initialized before a graph is used for the first time
2. We can use gradient methods to modify variables after each iteration as we search for a model's optimal parameter settings
3. 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 feedforward neural network.

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

Here we pass two arguments to `tf.Variable`. The first, `tf.random_normal`, 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 300x200, 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:

```
# 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:

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

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

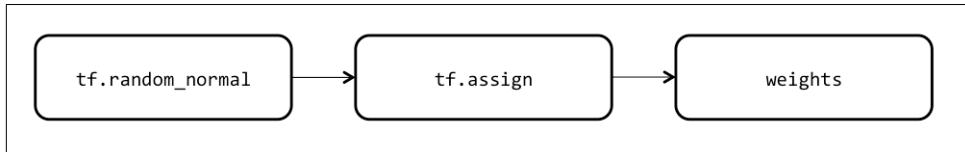


Figure 3-1. Three operations are added to the graph when instantiating a TensorFlow variable. In this example, we instantiate the variable `weights` using a random normal initializer.

As we mention above, before we use any TensorFlow variable, the `tf.assign` operation must be run so that the variable is appropriately initialized with the desired value. We can do this by running `tf.initialize_all_variables()`, which will trigger all of the `tf.assign` operations in our graph. We'll describe this in more detail when we discuss sessions in TensorFlow.

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 the input (adding tensors of type `float32` vs. `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 a table from the original TensorFlow white paper detailing the various categories of operations in TensorFlow.

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. We instead need a component that we populate every single time the computation graph is run.

TensorFlow solves this problem using a construct called a *placeholder*. 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 mini-batch 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 mini-batch 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*. The TensorFlow session is responsible for building the initial graph, 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)
```

The first 4 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.

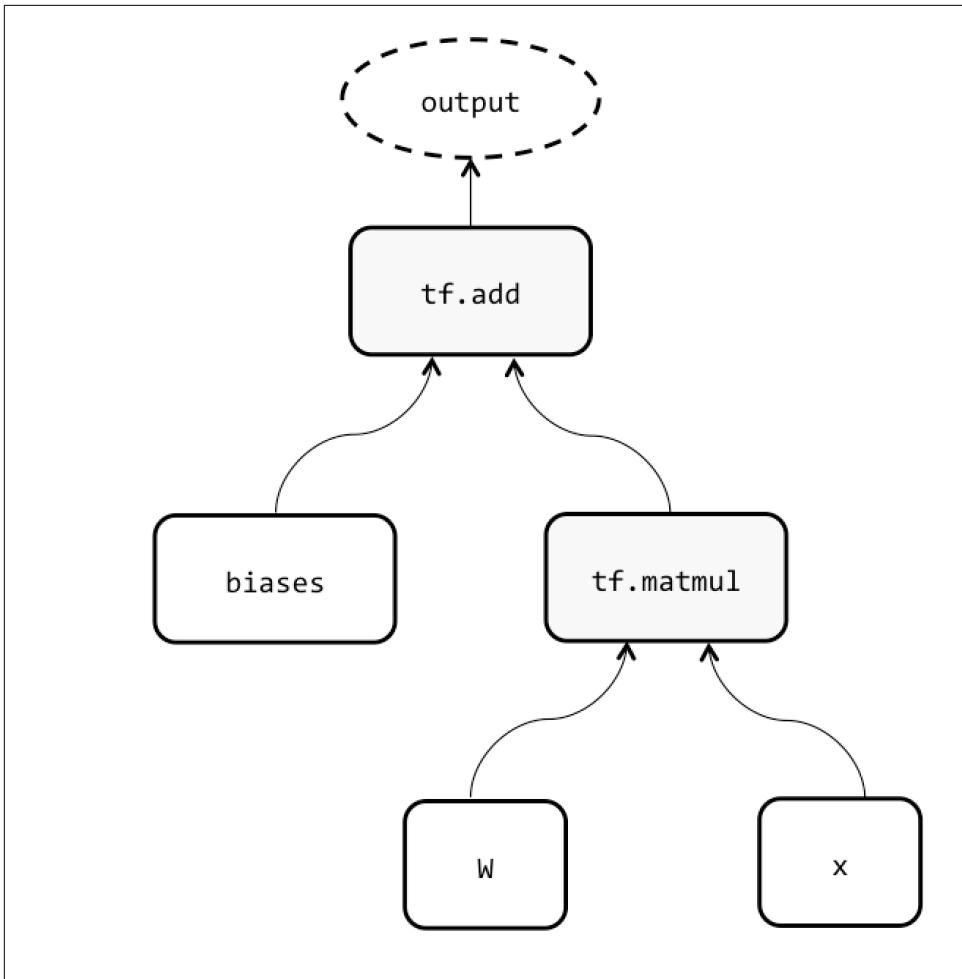


Figure 3-2. This is a an example of a simple computational graph in TensorFlow

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 before we jump into model training, 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 re-using 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 6 variables describing 3 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 instead, we want to reuse the model and its variables. It turns out, 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:

1. `tf.get_variable(<name>, <shape>, <initializer>)`: checks if a variable with this name exists, retrieves the variable if it does, creates it using the shape and initializer if it doesn't
2. `tf.variable_scope(<scope_name>)`: manages the namespace and determines the scope in which `tf.get_variable` operates

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`", etc.

```
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 above:

```
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 you 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 ID's and normally consist of the following:

1. `"/cpu:0"`: The CPU of our machine.
2. `"/gpu:0"`: The first GPU of our machine, if it has one.
3. `"/gpu:1"`: The second GPU of our machine, if it has one.
4. ... etc ...

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` 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:

```
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**. Sample code for multi-GPU code is shown below:

```
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)
```

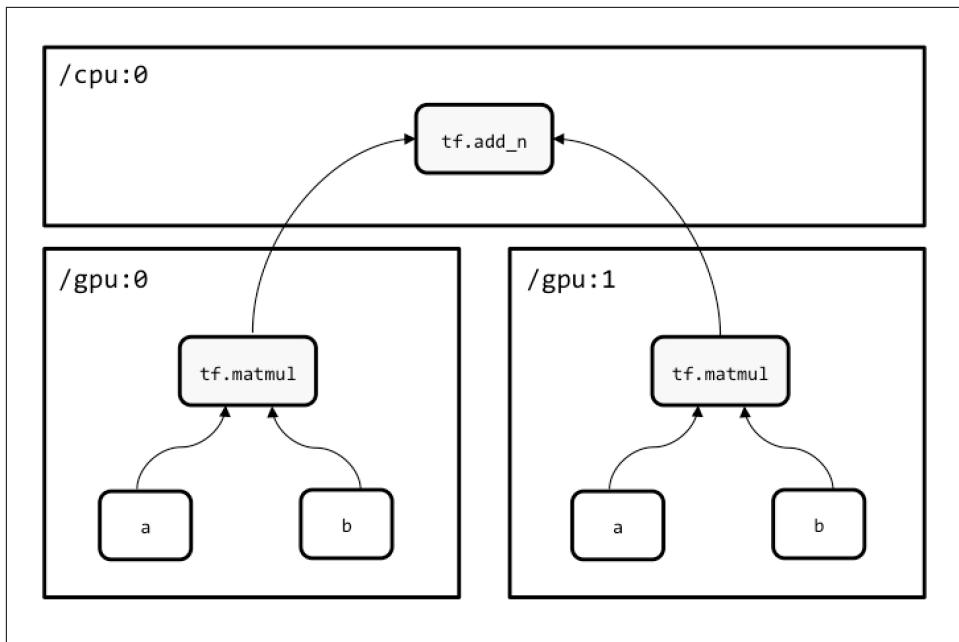


Figure 3-3. Building multi-GPU models in a tower-like fashion

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.

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 a input x belongs to class i using the softmax expression we talked about earlier:

$$P(y = i | x) = \text{softmax}_i(Wx + b) = \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}$$

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 below in **Figure 3-4** (bias connections not shown to reduce clutter).

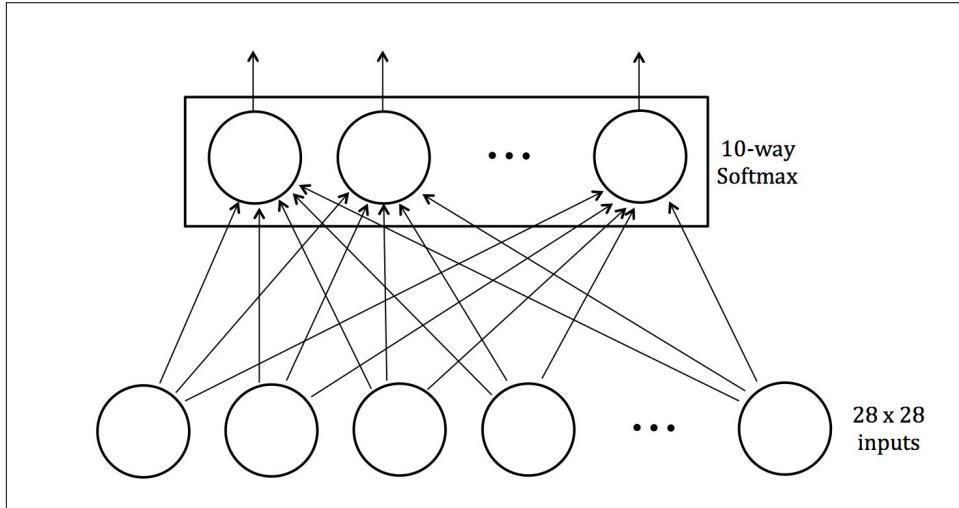


Figure 3-4. Interpreting logistic regression as a primitive neural network

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 a 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 towards 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**: which produces a probability distribution over the output classes given a minibatch
2. **loss**: which computes the value of the error function (in this case, the cross entropy loss)
3. **training**: which is responsible for computing the gradients of the model's parameters and updating the model

4. evaluate: which 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 mini-batches that has 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` and `tf.histogram_summary` commands to log the cost for each minibatch, validation error, and the distribution of parameters. For reference, we demonstrate scalar summary statistic for the cost function below:

```
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` 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.

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 5 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):
    minibatch_x, minibatch_y = mnist.train.next_batch(batch_size)
    # Fit training using batch data
    feed_dict = {x : minibatch_x, y : minibatch_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 feedforward 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. TensorBoard comes with a visuali-

zation tool called TensorBoard which provides an easy-to-use interface for navigating through our summary statistics. 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.

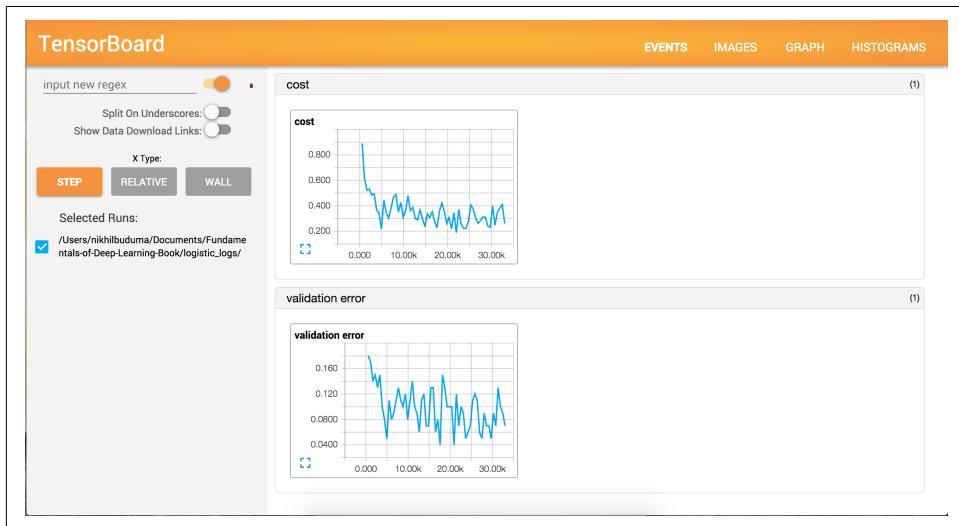


Figure 3-5. The TensorBoard events view

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

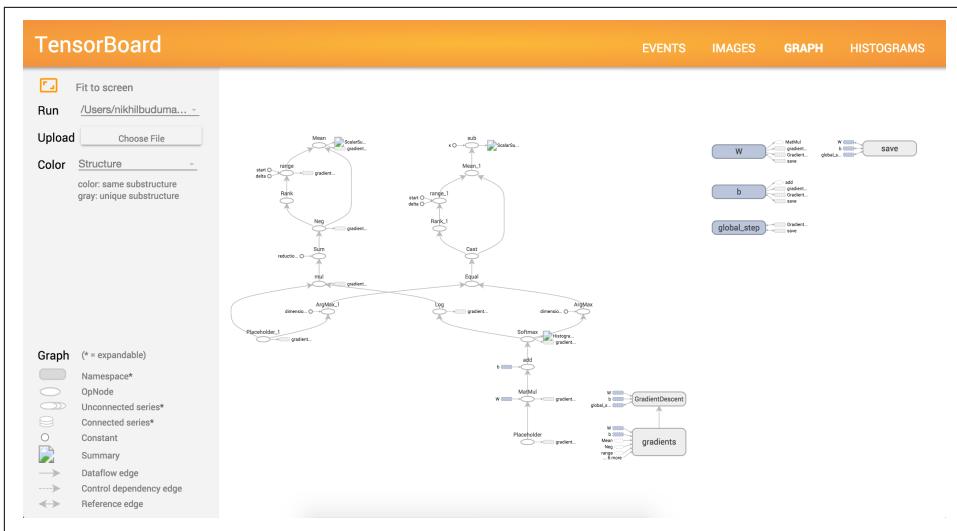


Figure 3-6. The TensorBoard graph view

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 (\$1000 to \$9999), 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 feedforward network to tackle the MNIST challenge.

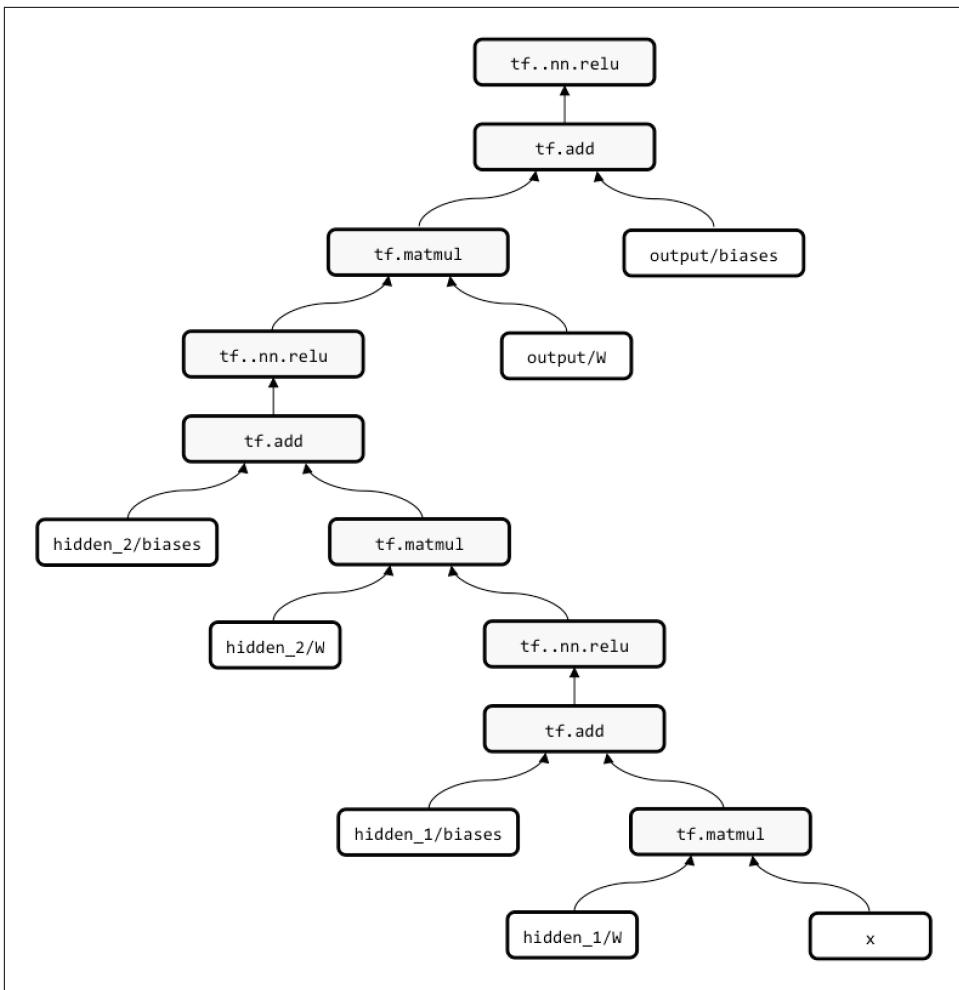


Figure 3-7. A feedforward network powered by ReLU neurons with two hidden layers

We construct a feedforward model with two hidden layers, each with 256 ReLU neurons as shown in **Figure 3-7**. 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 $\frac{2}{n_{in}}$, where n_{in} is the number inputs coming into the neuron. 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 modification below:

```

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.

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, and operations, computation graphs and devices. In the final sections, we used this understanding to train and visualize a logistic regression model and a feedforward neural network using stochastic gradient descent. Although the logistic network model made many errors on the MNIST dataset, our feedforward 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 section, we'll begin to grapple with many of the problems that arise as we begin to make our networks deeper. While deep models afford us the power to tackle more difficult problems, they are also notoriously difficult to train with vanilla stochastic gradient descent. 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 in to modern optimization theory and design better algorithms for training deep networks.

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. 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, in addition to the adoption of neurons with ReLU activations, breakthroughs in non-convex optimization 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 non-convex error surfaces induced by deep models, why vanilla mini-batch gradient descent falls short, and how modern non-convex optimizers overcome these pitfalls.

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 United States 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 United States is extremely complex, i.e. is a non-convex 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 overcome shallow local minima. But as we can see in **Figure 4-1**, even a stochastic error surface won't save us from a deep local minimum.

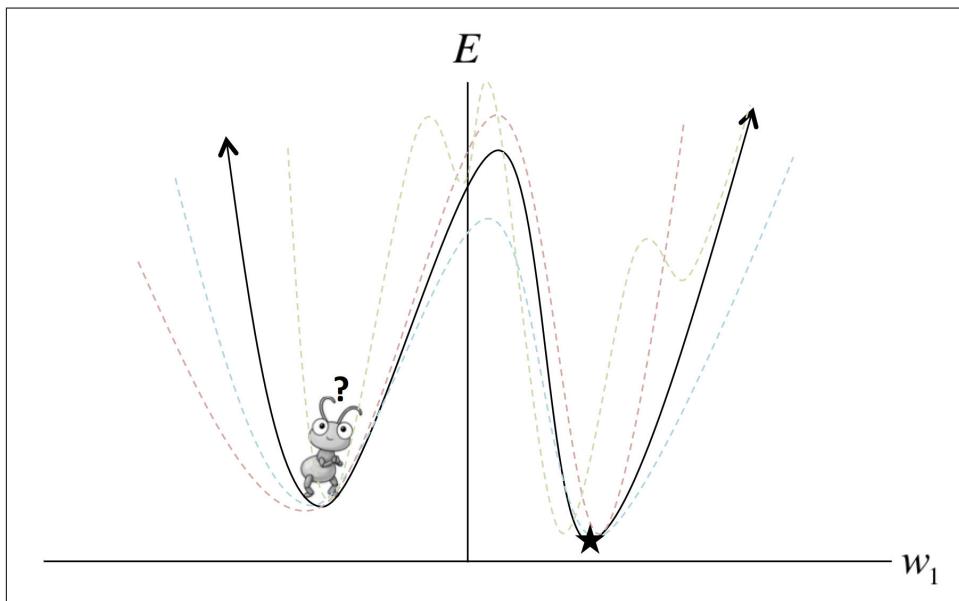


Figure 4-1. Mini-batch gradient descent may aid in escaping shallow local minima, but often fails when dealing with deep local minima as shown.

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 why this observation is true.

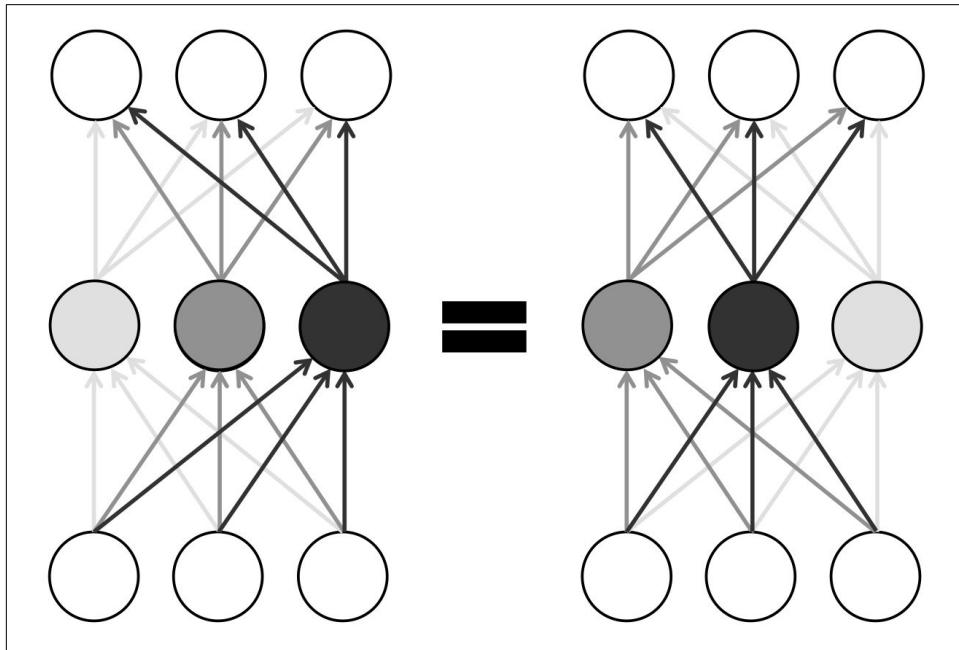


Figure 4-2. Rearranging neurons in a layer of a neural network results in equivalent configurations due to symmetry.

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 3-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.

In addition to the symmetries of neuron re-arrangements, non-identifiability is also present in other forms in certain kinds of neural networks. For example, there are infinite number of equivalent configurations that for an individual ReLU neuron that result in equivalent networks. Because an ReLU uses a piece-wise 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 non-identifiable 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 minima 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 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 2015 that attempted to separate these two potential confounding factors. Instead of analyzing the error function over time, they cleverly investigated what happens on the error surface between an randomly initialized parameter vector and a successful final solution by using linear interpolation. So given a randomly initialized parameter vector θ_i and 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 showed 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 feedforward ReLU network we built in the previous chapter. Using a checkpoint file that we saved while training our original feedforward 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** below, 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 in two sections.

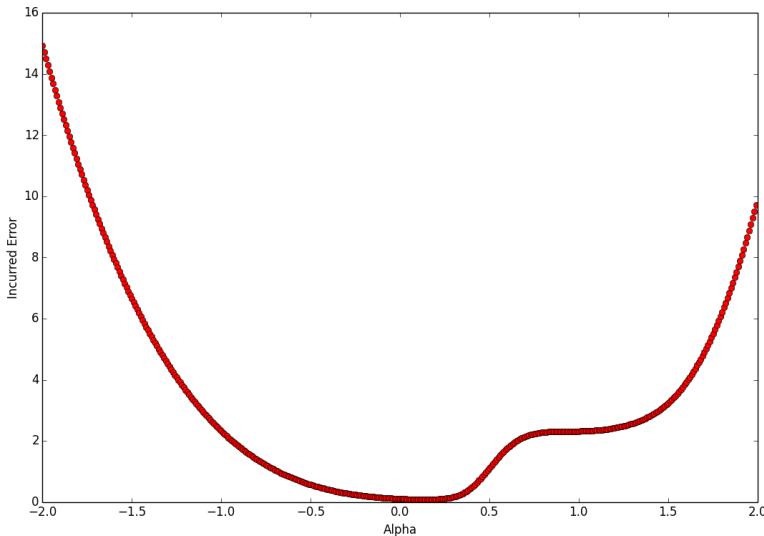


Figure 4-3. The cost function of a 3-layer feedforward network as we linearly interpolate on the line connecting a randomly initialized parameter vector and an SGD solution.

Flat Regions in the Error Surface

Although it seems that our analysis is devoid of troublesome local minima, 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 intuitively understand why.

For a one dimensional cost function, a critical point can take one of 3 forms, as shown in **Figure 4-4**. Loosely, let's assume each of these 3 configurations is equally likely. This means given a random critical point in a random one-dimensional function, it has $1/3$ 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.

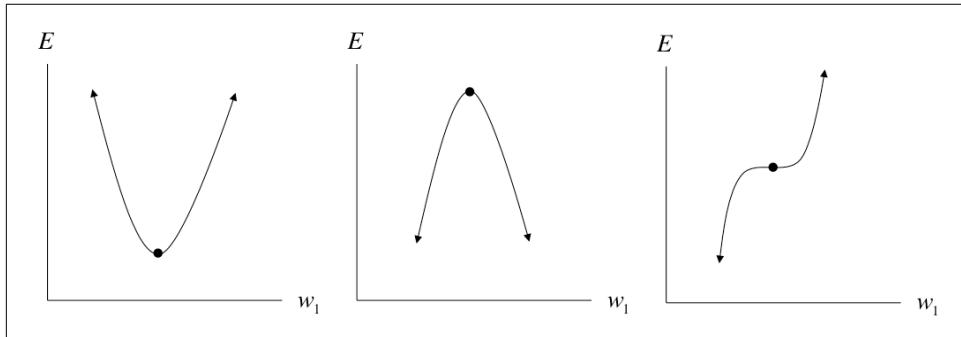


Figure 4-4. Analyzing a critical point along a single dimension

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 figuring out if this point is a local minima, local maxima, 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 minima or a maxima. In reality, it's neither. It's a more complex type of saddle point.

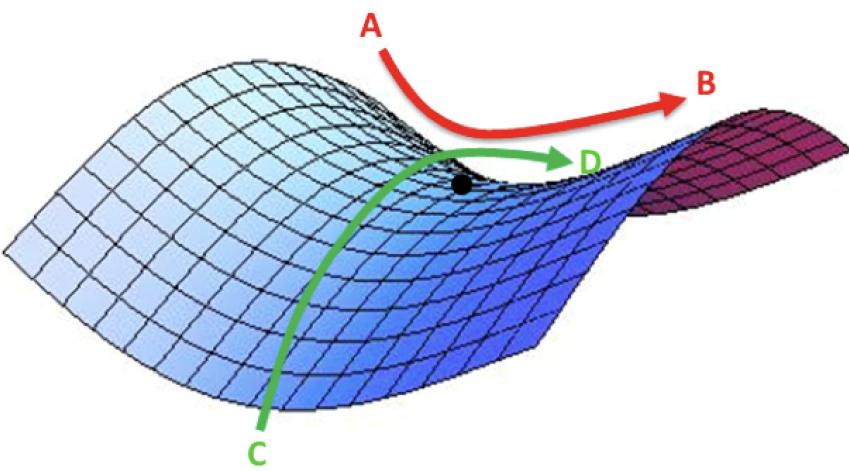


Figure 4-5. A saddle point over a 2 dimensional error surface.

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 3 different flavors in a one-dimensional subspace, we realize that the probability that a random critical point in a random function is $\frac{1}{3^d}$. This means that a random 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.

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 in a later section.

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**.

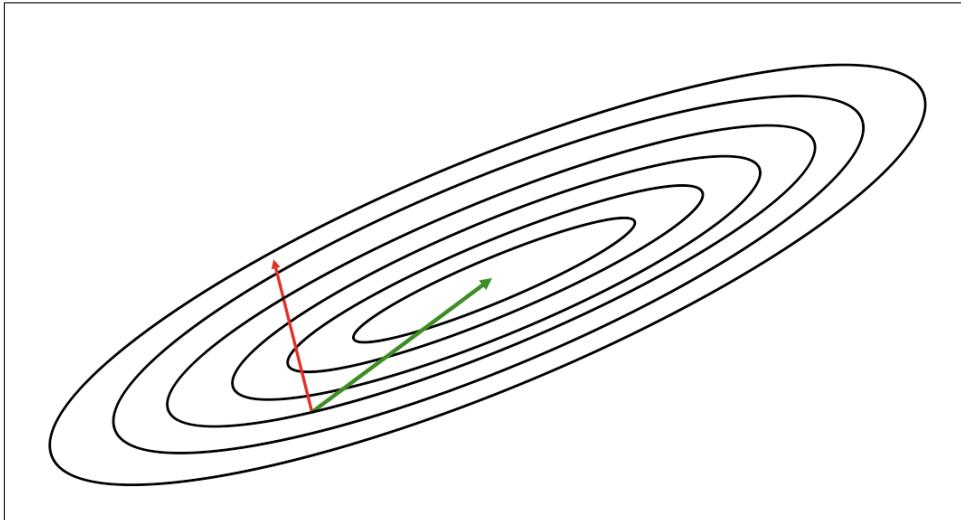


Figure 4-6. Local information encoded by the gradient usually does not corroborate the global structure of the error surface.

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.

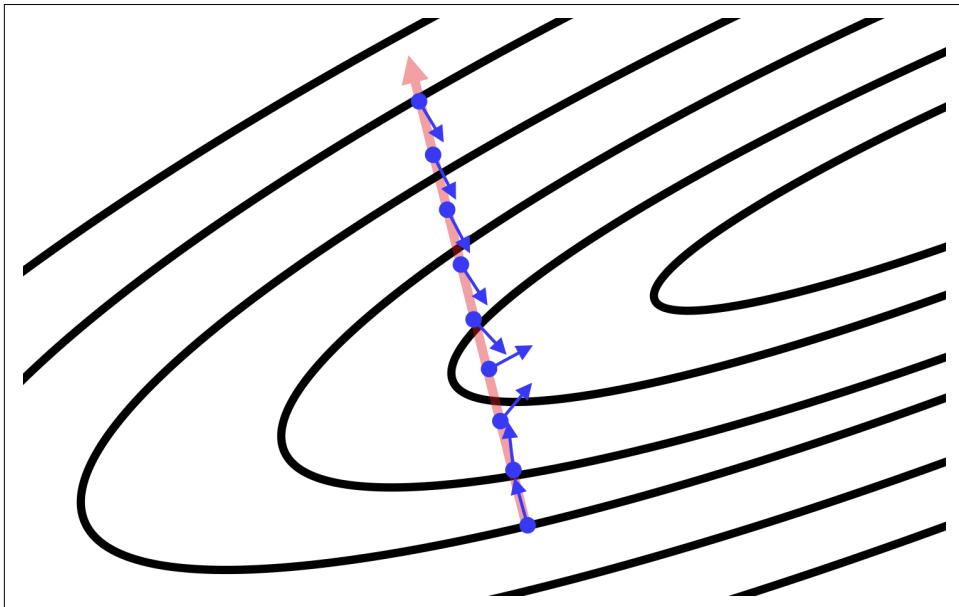


Figure 4-7. We show how the direction of the gradient changes as we move along the direction of steepest descent (as determined from a starting point). The gradient vectors are normalized to identical length to emphasize the change in direction of the gradient vector.

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 $\frac{\partial(\partial E/\partial w_j)}{\partial w_i}$, 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* \mathbf{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 (specifically that the Hessian matrix is real and symmetric) of the Hessian matrix 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 \mathbf{d} , the second derivative in that direction is given by $\mathbf{d}^\top \mathbf{H} \mathbf{d}$. 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:

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

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 minima 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 acceleration. But acceleration does not singlehandedly 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 hyper parameter 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*. 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 parti-

cle 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 shown below:

```
step_range = 10
step_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 below:

```
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. an 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 4-7**.

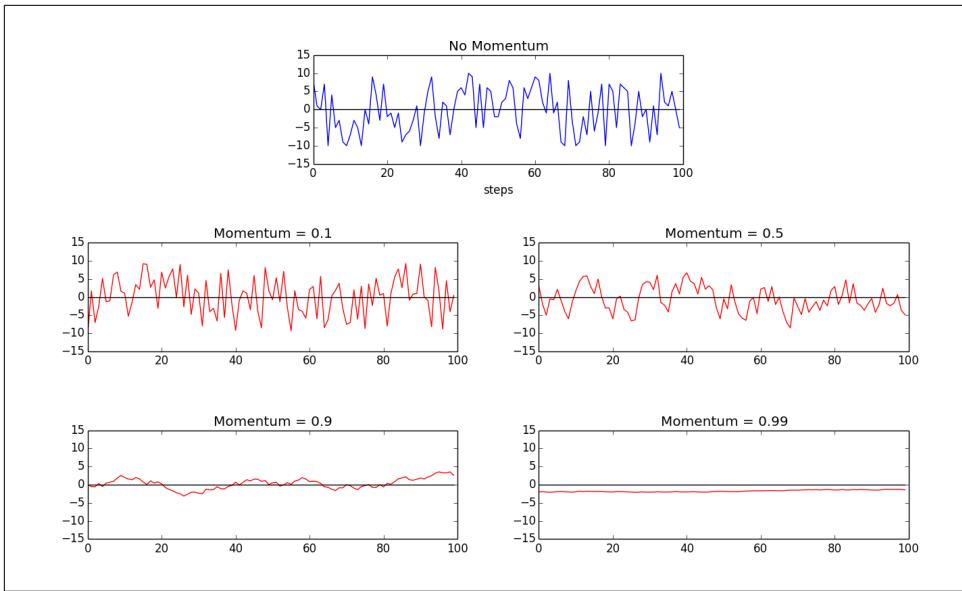


Figure 4-8. Momentum smooths volatility in the step sizes during a random walk using an exponentially weighted moving average

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 4-8**. 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.

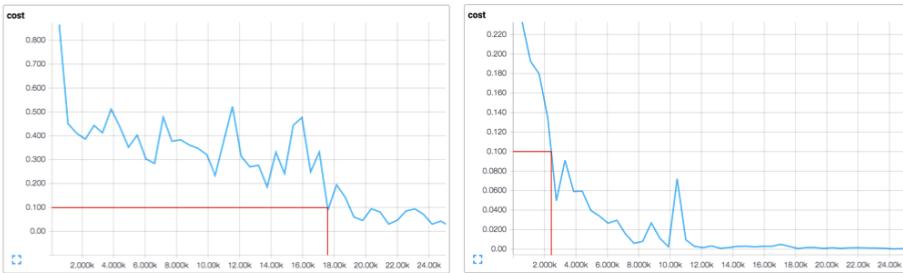


Figure 4-9. Comparing training a feedforward network with (right) and without (left) momentum demonstrates a massive decrease in training time

Recently, more work has been done exploring how the classical momentum technique can be improved. Sutskever et al. in 2013 propose an alternative called Nesterov momentum that computes the gradient on the error surface at $\theta + \mathbf{v}_{i-1}$ during the velocity update instead of at θ . 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 minibatch gradient descent used in most deep learning optimization approaches. Support for Nesterov momentum is not yet available in TensorFlow as of the writing of this text.

A Brief View of Second Order Methods

As we discussed above, 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 zig-zagging a significant amount as shown in **Figure 4-9** because each time we move in the direction of steep-

est 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 non-convex error surfaces we find in deep networks.

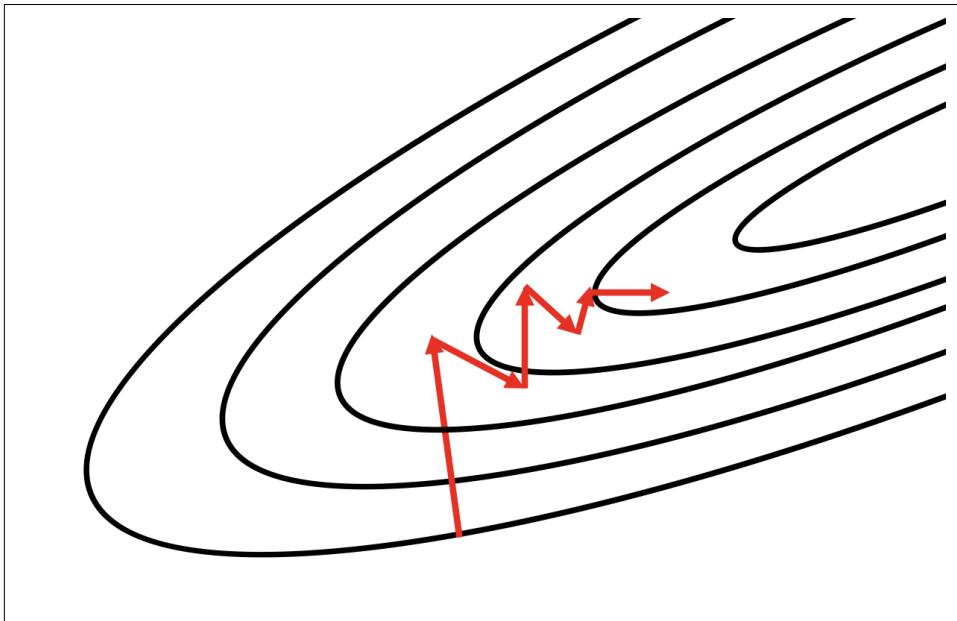


Figure 4-10. The method of steepest descent often zig-zags. Conjugate descent attempts to remedy this issue

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. In its original form, BFGS has a significant memory footprint, but recent work has produced a more memory efficient version known as *L-BFGS*.

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 doesn't learn quickly enough, but a learning rate that is too large may have difficulty converging as we approach a local minima 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. 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 broadcasted 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.

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 rate. 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.

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,
                          momentum=0.0, epsilon=1e-10,
                          use_locking=False, name='RMSProp')
```

As the template suggests, we can utilize RMSProp with momentum (specifically Nesterov 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.

Adam - Combining Momentum and RMSProp

Before concluding our discussion of modern optimizers, we discuss one final algorithm - Adam. 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:

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

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

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

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 $\mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i]$.

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

We can also assume that $\mathbb{E}[\mathbf{g}_k \odot \mathbf{g}_k] \approx \mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i]$, 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 ():

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

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

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

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

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

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

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

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 their 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 minibatch gradient descent, minibatch 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 feedforward network model we built.

```
$ python optimizer_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 the text.

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 spurious local minima may likely be exaggerated, saddle points and ill-conditioning do pose a serious threat to the success of vanilla minibatch 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.

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. But not only can we name objects we are looking at, but 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.

Foundational to the human sense of vision is the neuron. Specialized neurons are responsible for capturing light information in the human eye. This light information is then pre-processed, 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.

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.

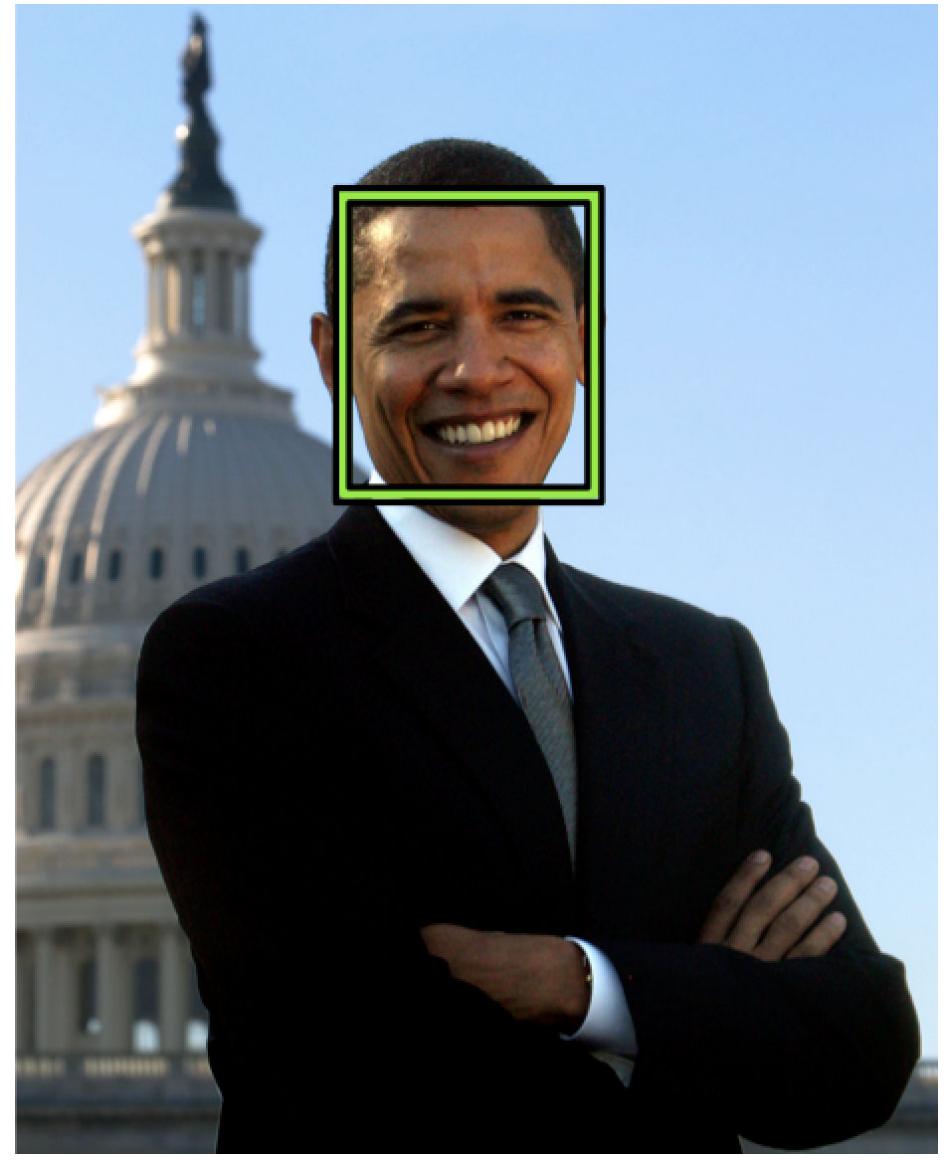


Figure 5-1. A hypothetical face-recognition algorithm should detect a face in this photograph of President Obama

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 first chapter) by giving it the raw pixel values of the image and hop-

ing 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 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**.

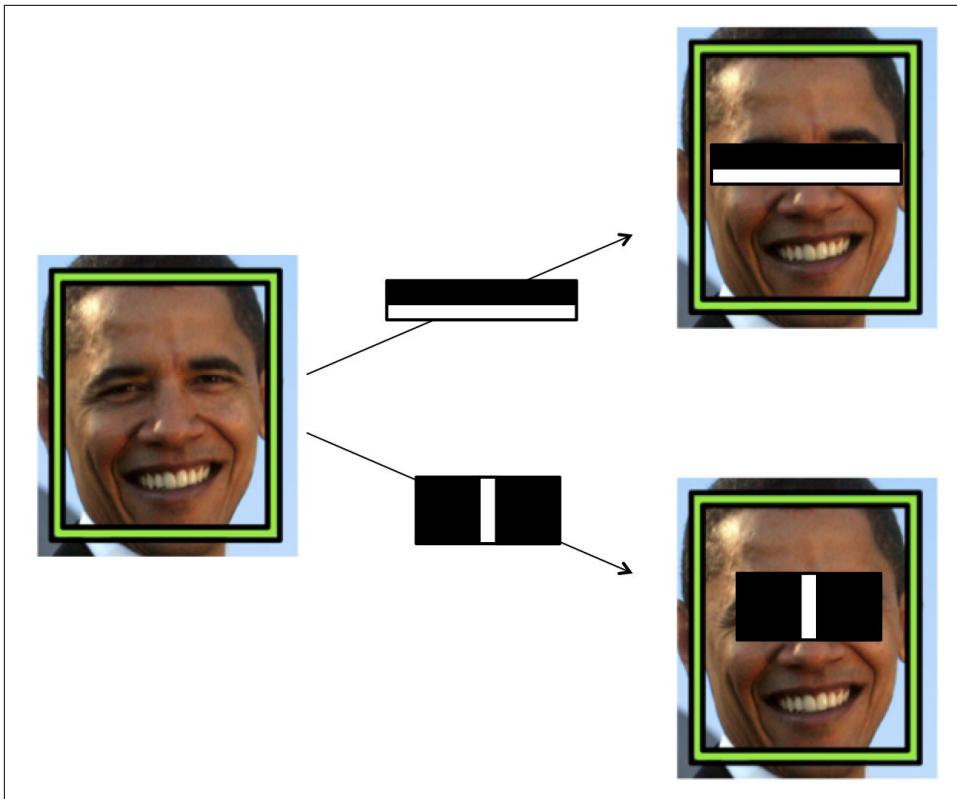


Figure 5-2. An illustration of Viola-Jones intensity detectors

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 [here](#)), 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 were fundamental limitations of the algorithm. If a face is partially covered with shade, the light intensity comparisons no longer work. Moreover, if algorithm were 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). 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 5 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. 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 error rate of approximately 16%. It would be no understatement to say that AlexNet singlehandedly 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.

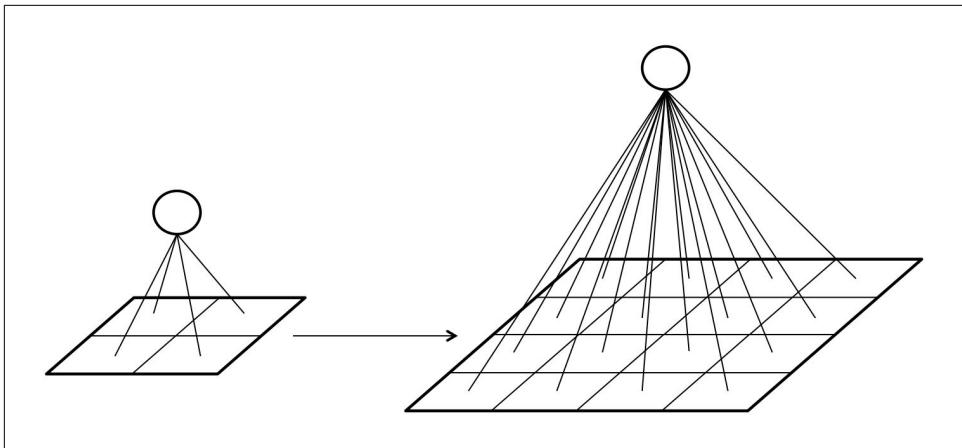


Figure 5-3. The density of connections between layers increases intractably as the size of the image increases

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 by 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 by 200 pixel image, our input layer would have $200 \times 200 \times 3 = 120,000$ 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.

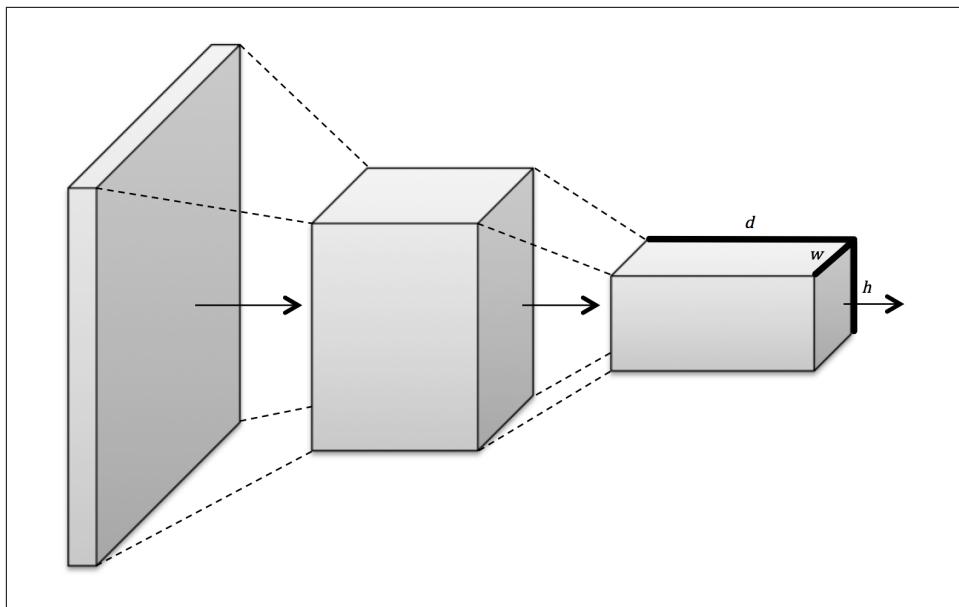


Figure 5-4. Convolutional layers arrange neurons in 3-dimensions, so layers have width, height, and depth

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**. 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 neuron. A convolutional layer's function can be expressed simply: it processes a 3-dimensional volume of information to produce a new 3-dimensional volume of information. We'll take a closer look at how this works in the next section.

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

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**.

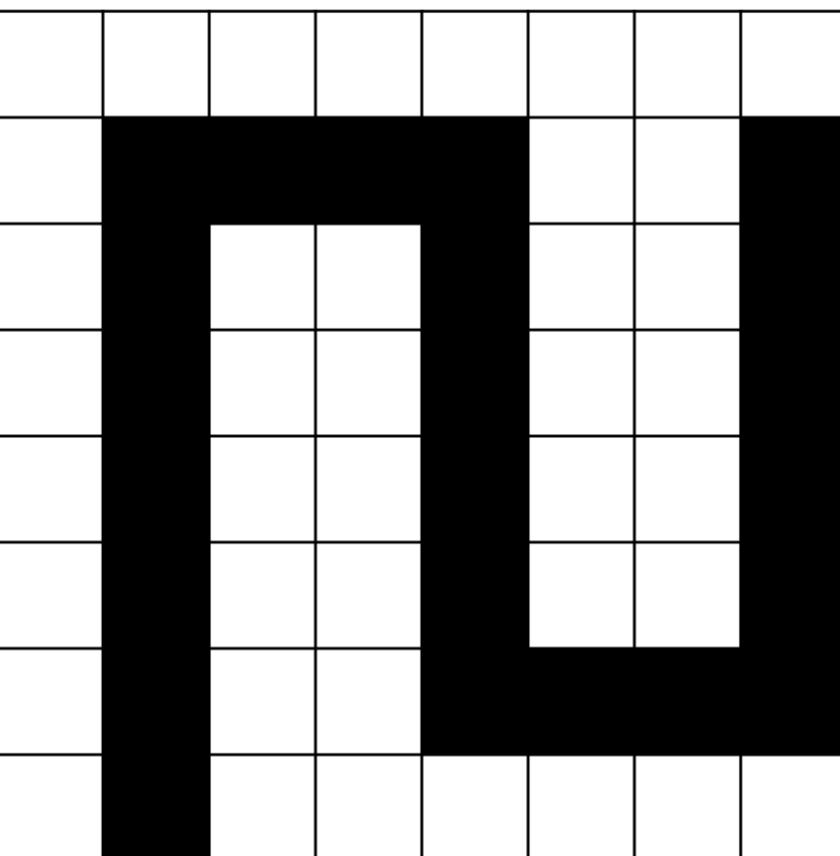


Figure 5-5. We'll analyze this simple black-and-white image as a toy example

Let's say that we want to use 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, and 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.

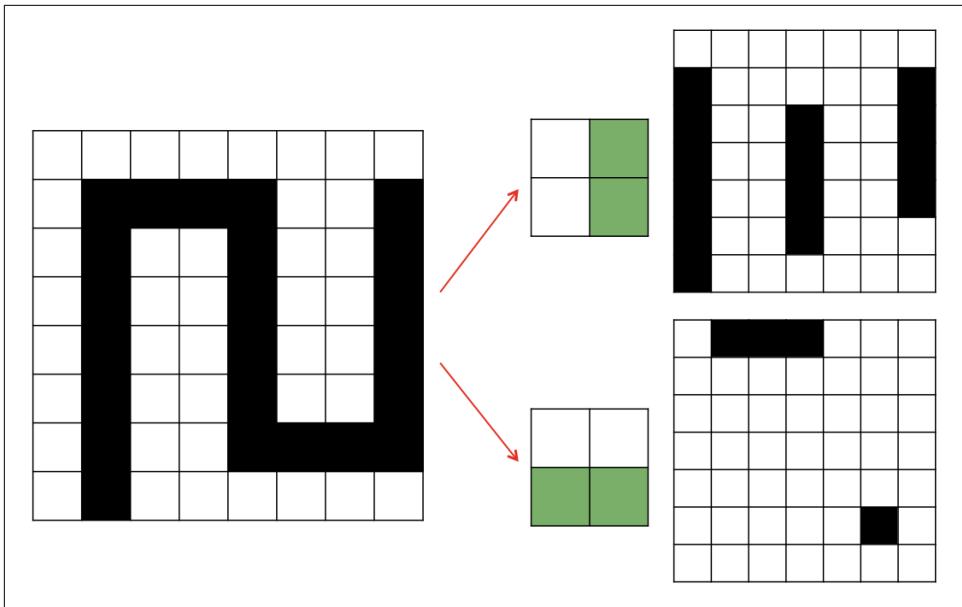


Figure 5-6. Applying filters that detect vertical and horizontal lines on our toy example

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.

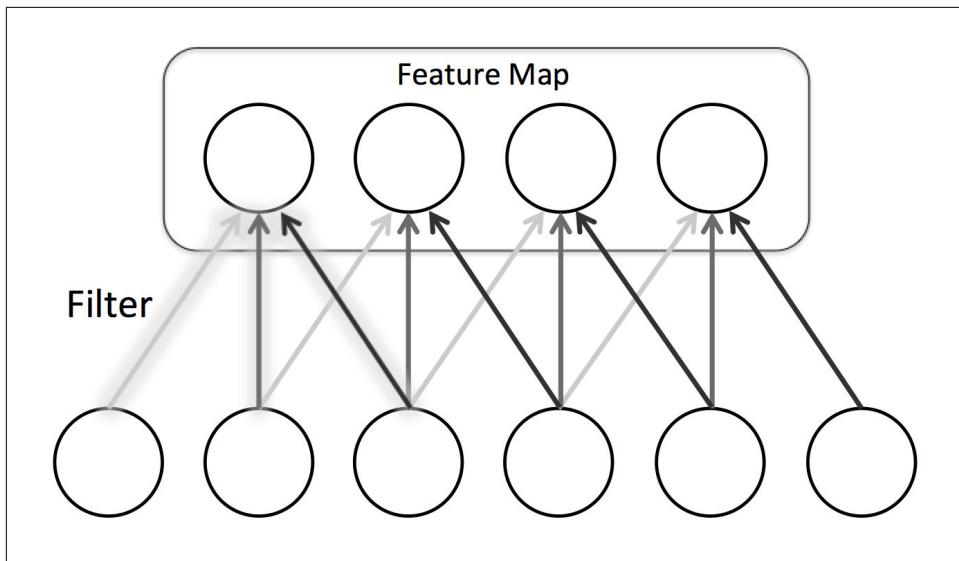


Figure 5-7. Representing filters and feature maps as neurons in a convolutional layer

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 kept identical for all of the neurons in a feature map), we can mathematically express the feature map as follows:

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

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.

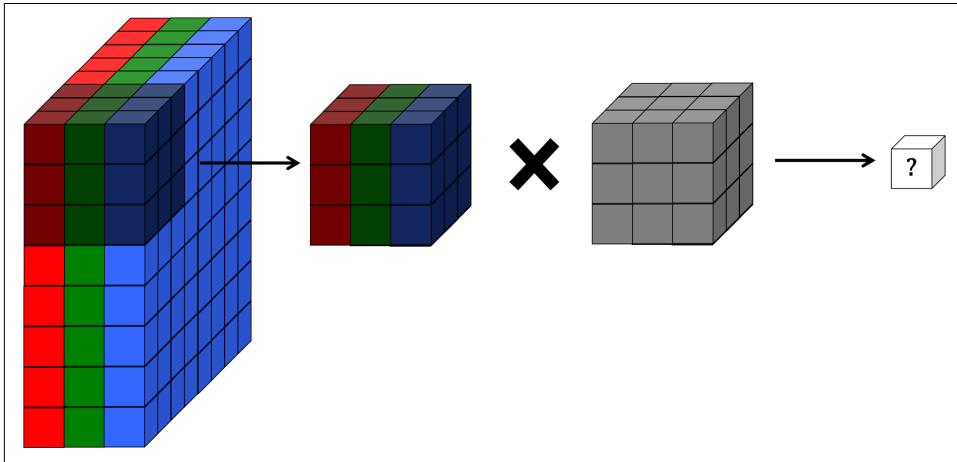


Figure 5-8. Representing a full color RGB image as a volume and applying a volumetric convolutional filter

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 the number of filters in that layer, because each filter produces its own slice. We visualize these relationships in **Figure 5-9**.

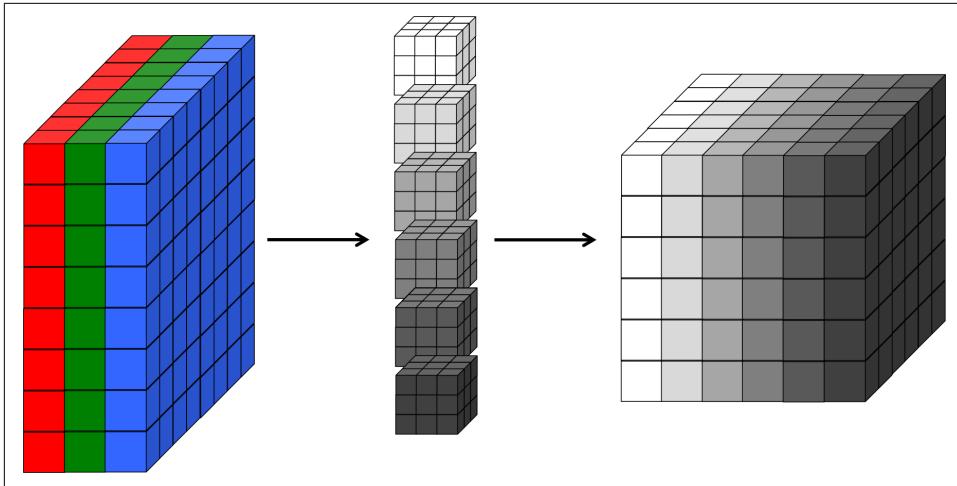


Figure 5-9. A 3-dimensional visualization of a convolutional layer, where each filter corresponds to a slice in the resulting output volume.

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:

- Their *width* w_{in}
- Their *height* h_{in}
- Their *depth* d_{in}
- Their *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 filter's height and width
- Their *stride* s , or the distance between consecutive applications of the filter on the input volume. If $s = e$, we'd 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

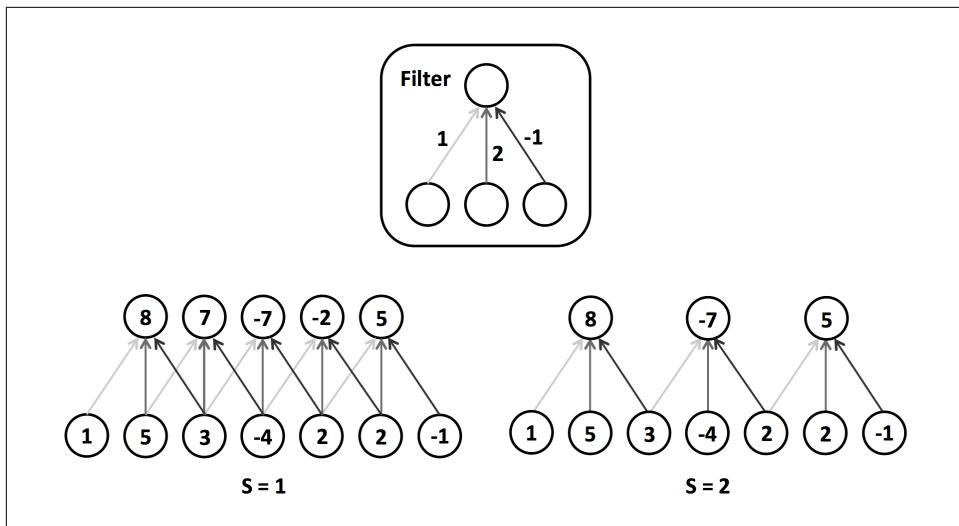


Figure 5-10. An illustration of a filter's stride hyperparameter

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 $5 \times 5 \times 3$ input volume with zero padding $p = 1$. We'll use two $3 \times 3 \times 3$ filters (spatial extent) with a stride $s = 2$. We'll use a linear function to produce the output volume, which will be of size $3 \times 3 \times 2$.

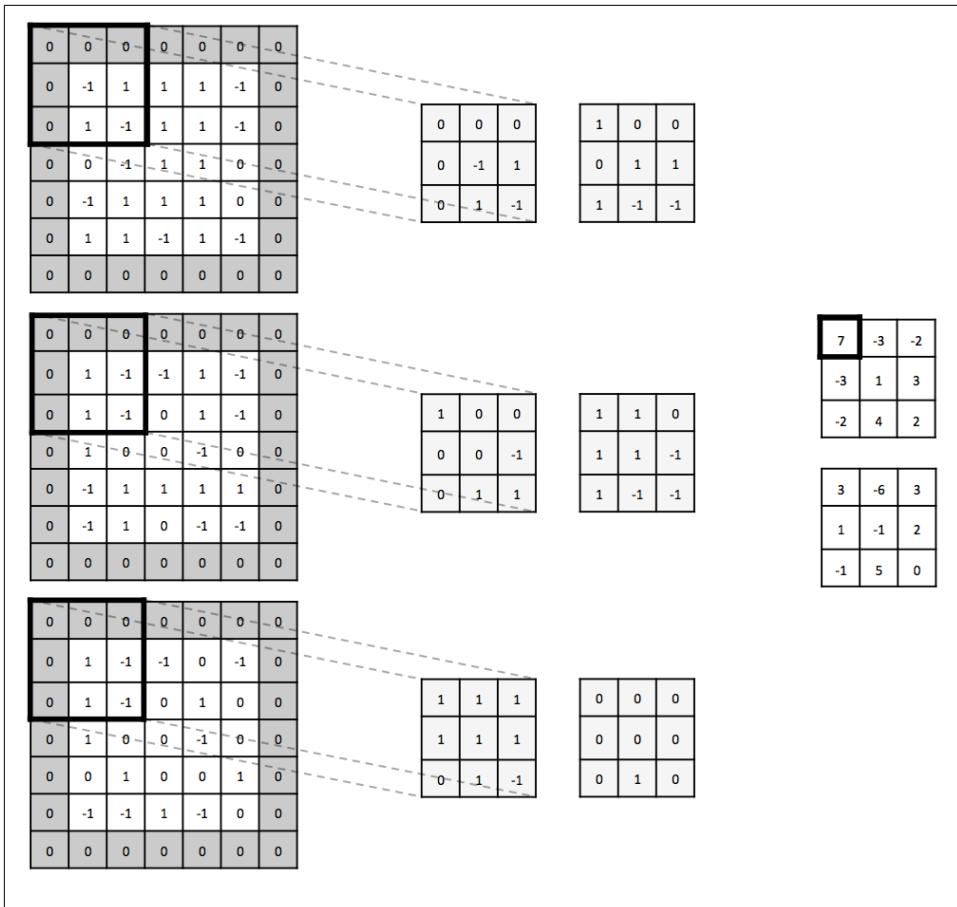


Figure 5-11. This is a convolutional layer with an input volume that has width 5, height 5, depth 3, and zero padding 1. There are 2 filters, with spatial extent 3 and applied with a stride of 2. It results in an output volume with width 3, height 3, and depth 2. We apply the first convolutional filter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

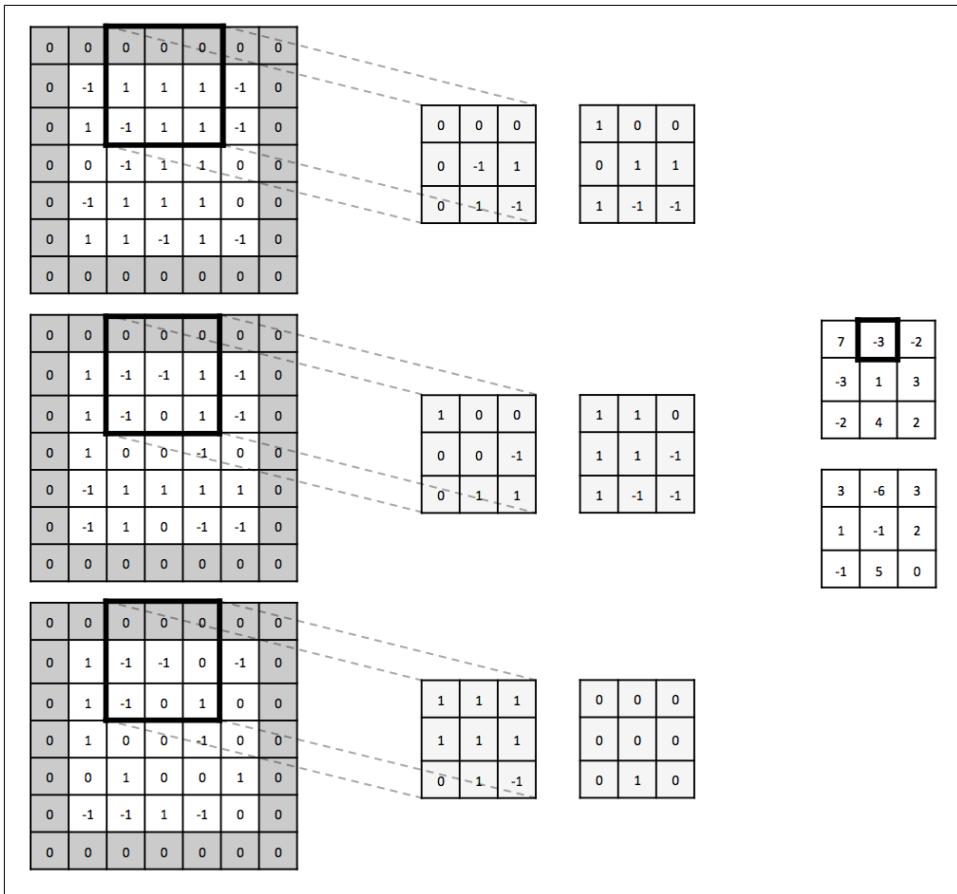


Figure 5-12. Using the same setup as Figure 5-11, we generate the next value in the first depth slice of the output volume.

Generally, it's wise to keep filter sizes small (size 3x3 or 5x5). Less commonly, larger sizes are used (7x7) 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):

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

Here, `input` is a 4-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 4-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 is 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. 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** below:

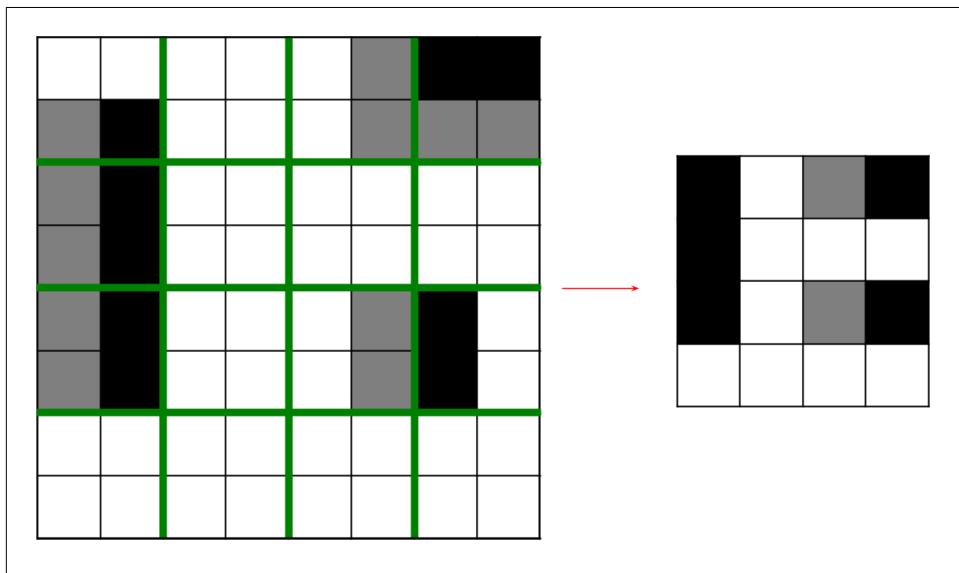


Figure 5-13. An illustration of how max pooling significantly reduces parameters as we move up the network

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

- Its spatial extent e
- Its stride s

It's important to note that only two major variations of the pooling layer are used. The first is the non-overlapping 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 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 from Graham out of the University of Warwick, who proposes a concept called *fractional max pooling*. In fractional max pooling, a pseudorandom number generator is used to generate tilings with non-integer 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.

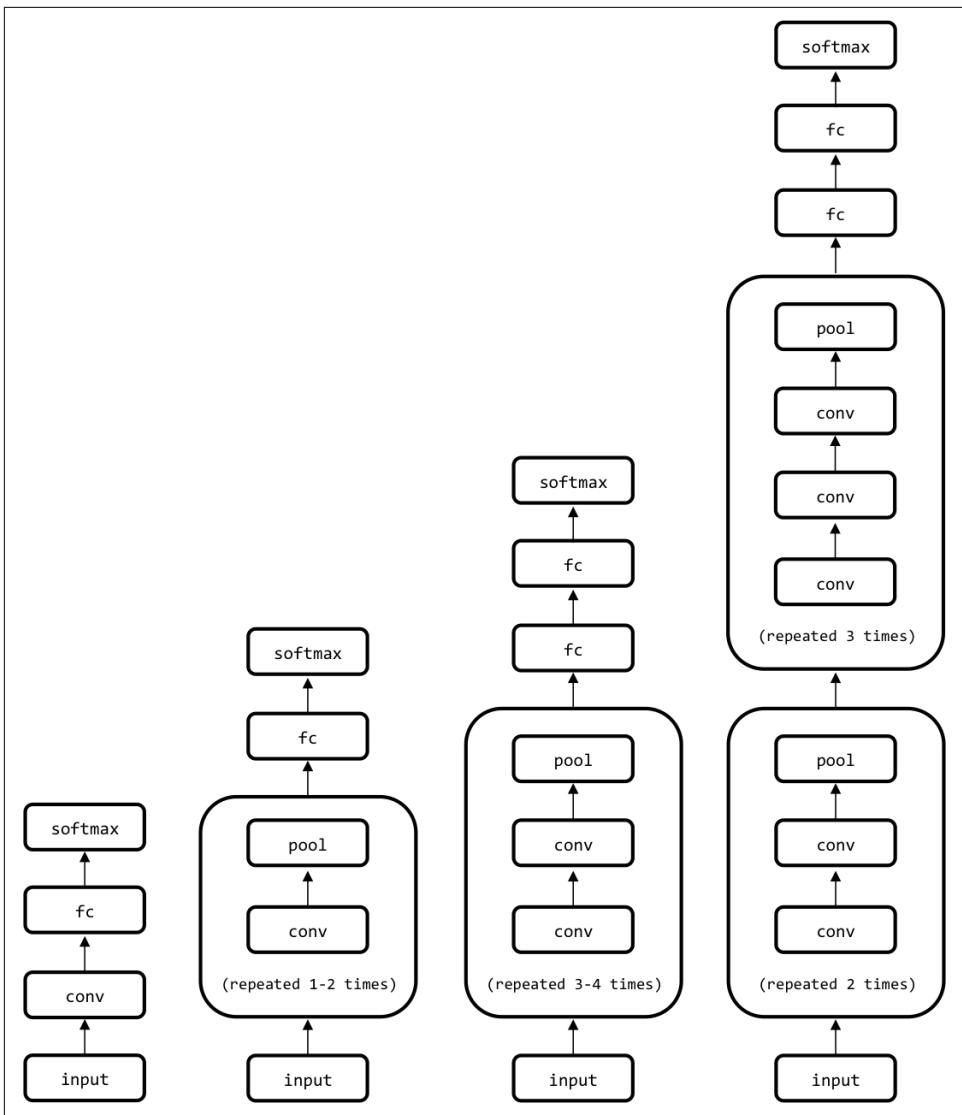


Figure 5-14.

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 memory 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. 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 feedforward network was able to achieve a 98.2% accuracy. Our goal will be to push the envelop 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 feedforward 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 feedforward 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.

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 convolu-

tional 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 1024. 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 per 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 99.7-99.8%), 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’s no color aberrations because MNIST is black and white, etc. 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 help 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 feedforward and convolutional neural networks using a technique called *batch normalization*. We can think of the intuition behind batch normalization like a tower of blocks, as shown in **Figure 5-15**.

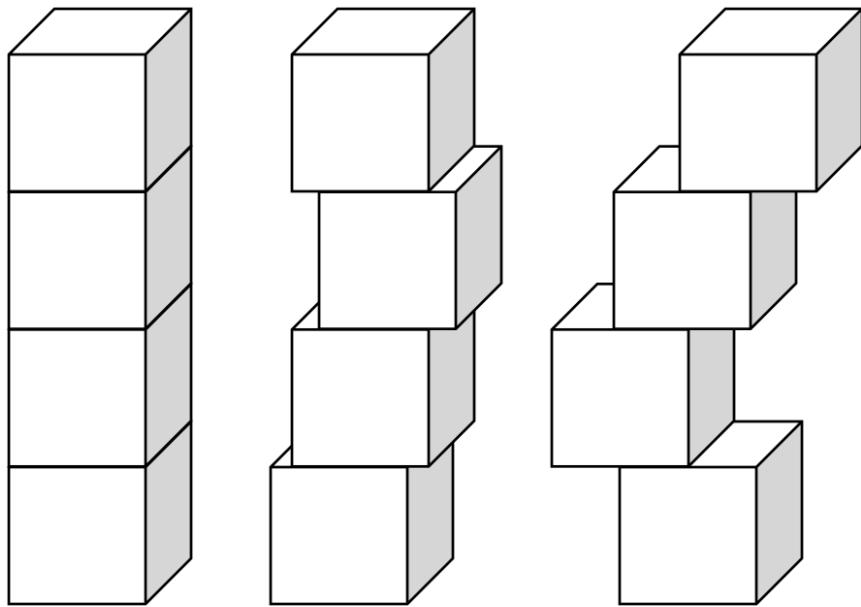


Figure 5-15. When blocks in a tower become shifted too drastically so that they no longer align, the structure can become very unstable.

When a tower blocks is stacked together neatly, the structure is stable. However, if we randomly shifted 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 begin 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.

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 mini-batch 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{x} , use an affine transform to restore representational power with two vectors of (trainable) parameters: $\gamma\hat{x} + \beta$

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 non-convolutional 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`. The code is shown below:

```
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 by 32 color images that belong to one of 10 possible classes. 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 by 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 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 in 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 in order to visualize training is plot the cost function and validation errors over time as training progresses. We can clearly demonstrate the benefits 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**.

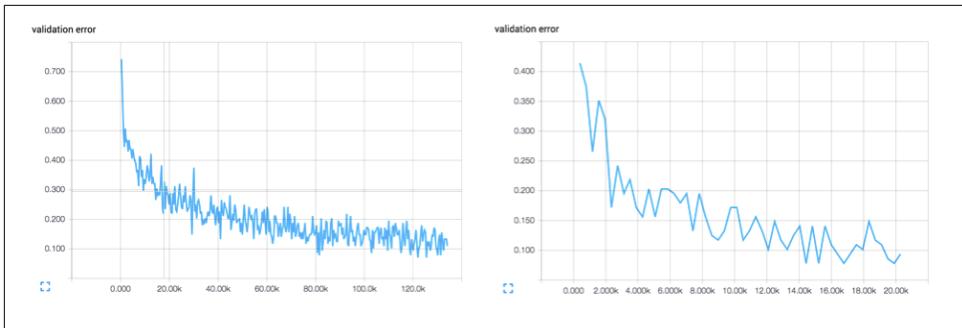


Figure 5-17. Training a convolutional network without batch normalization (left) vs. with batch normalization (right). Batch normalization vastly accelerates the training process.

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

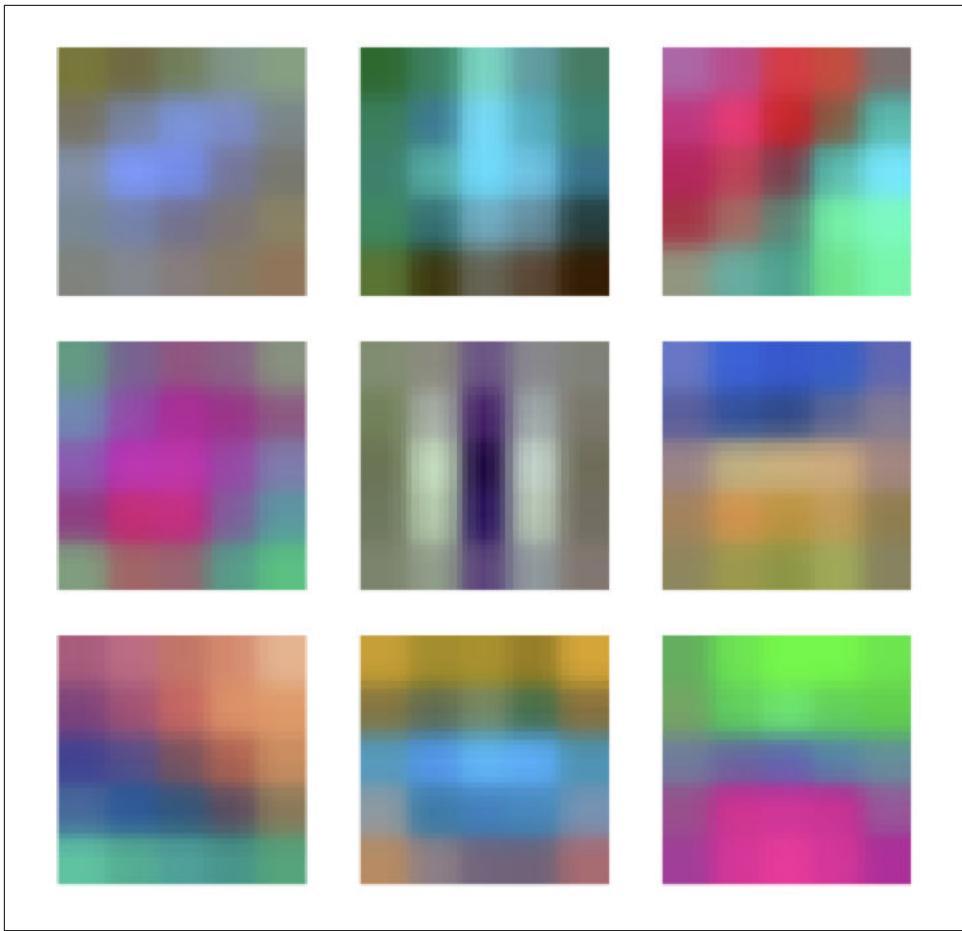


Figure 5-18. A subset of the learned filters in the first convolutional layer of our network

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 Neigh-*

bor Embedding or t-SNE to compress it to a 2-dimensional representation that we can visualize. We don't cover the details of t-SNE here, but there are a number of publicly available software tools will do it for us, including the script here: https://lvdmaaten.github.io/tsne/code/tsne_python.zip. 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

At first on a high level, it seems that images that are similarly colored to each other 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 once 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.

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*. The goal of neural style is to be able to take an arbitrary photograph and re-render 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 \mathbf{p} and the source of style \mathbf{a} . The third image is the generated image \mathbf{x} . Our goal will be to derive an error function that we can backpropagate through 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_p , 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 $\mathbf{F}^{(l)}$ of size $k_l \times m_p$. We can also represent all the activations of the photograph in a matrix $\mathbf{P}^{(l)}$ and all the activations of the generated image in the matrix $\mathbf{X}^{(l)}$. We use the `relu4_2` of the original VGGNet.

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

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 $k_l \times k_p$ for a given image is done as follows:

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

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

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

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`, `relu2_1`, `relu3_1`, `relu4_1` and `relu5_1` layers of the original VGGNet. We omit full a discussion of the TensorFlow code (https://github.com/dark-sigma/Fundamentals-of-Deep-Learning-Book/tree/master/neural_style) for brevity, but the results, as shown in **Figure 5-20**, are 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 5-dimensional tensors (including time as a dimension) and applying 3-dimensional convolutions is an easy way to extend the convolutional paradigm to video. Convolutional filters have also been successfully used to analyze audiograms. 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 teaching algorithms to play board games and analyzing bio-

logical 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.

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 labelled data is scarce but wild, unlabelled 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**.

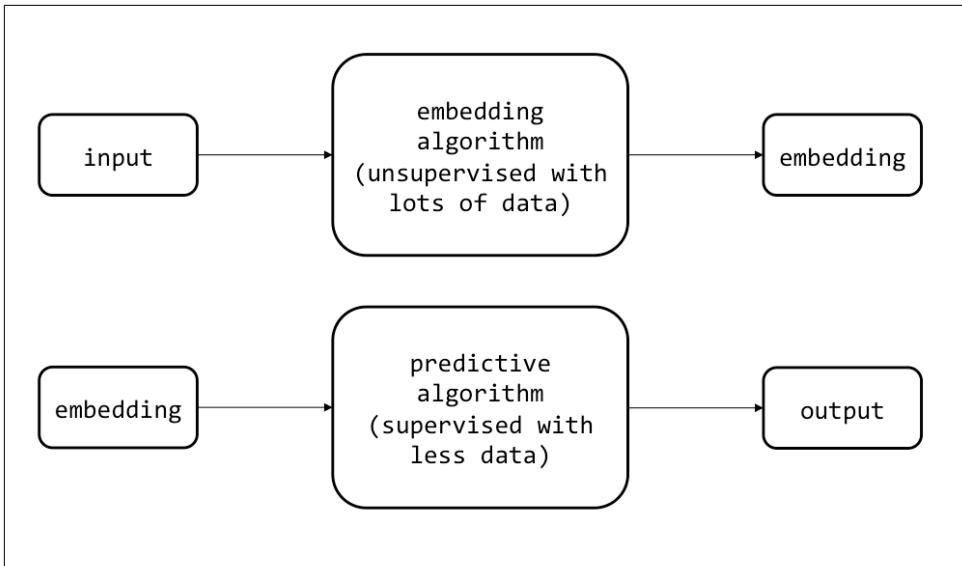


Figure 6-1. Using embeddings to automate feature selection in the face of scarce labelled data.

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 per-

form 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 that 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 the **Figure 6-2** below:

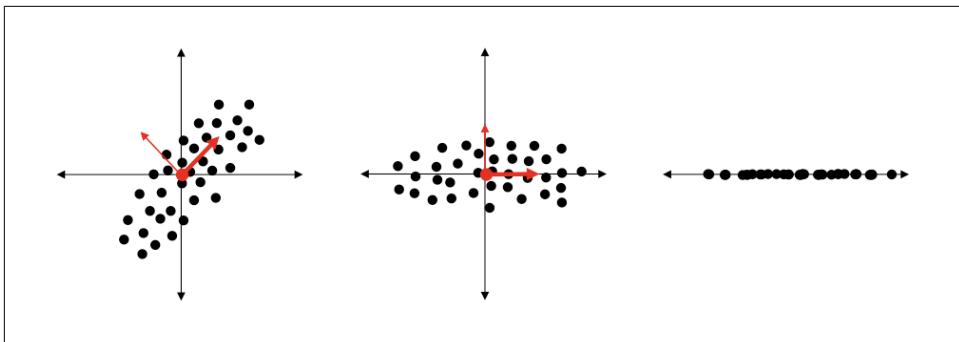


Figure 6-2. An illustration of PCA for dimensionality reduction to capture the dimension with the most information (as proxied by variance)

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 $\mathbf{T} = \mathbf{X}^{\top} \mathbf{W}$, where each column of \mathbf{W} corresponds to a eigenvector of the matrix $\mathbf{X}^{\top} \mathbf{X}$.

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

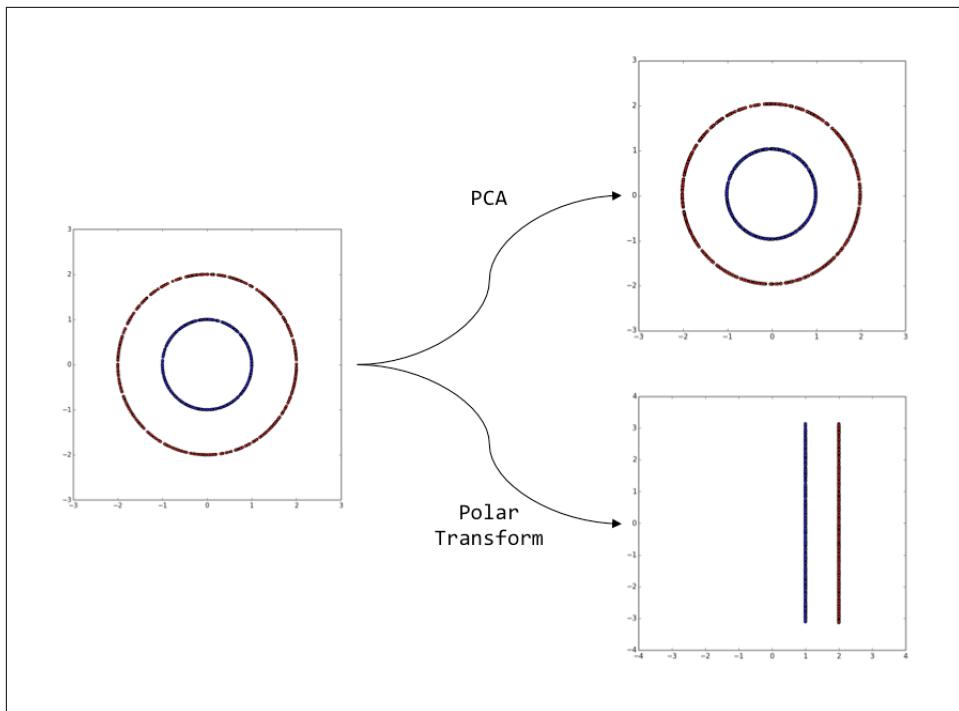


Figure 6-3. A situation in which PCA fails to optimally transform the data for dimensionality reduction.

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 non-linear relationships, we must develop a theory that will perform non-linear 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 feedforward networks, we discussed how each layer learned progressively more relevant representations of the input. In fact, in the previous chapter, 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 then compress the input 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 feedforward 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**.

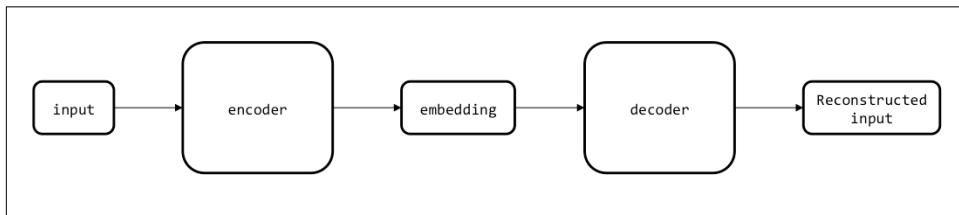


Figure 6-4. The autoencoder architecture attempts to construct a high dimensional input into a low dimensional embedding and then uses that low dimensional embedded to reconstruct the input.

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 describing the autoencoder was written by Hinton and Salakhutdinov in 2006. 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 section, we will recreate their experimental set up to validate this hypothesis and further explore the architecture and properties of feedforward autoencoders.

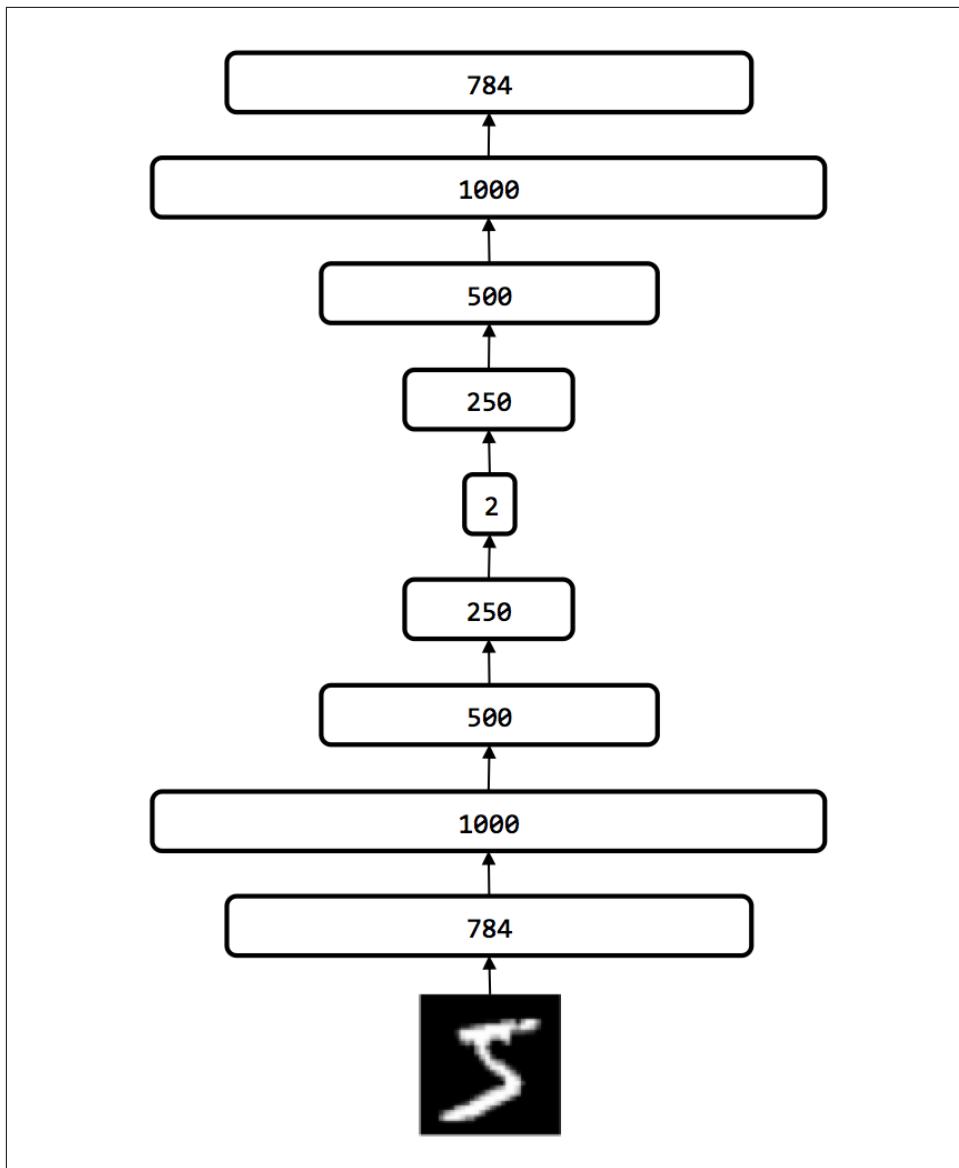


Figure 6-5. The experimental setup for dimensionality reduction of the MNIST dataset employed by Hinton and Salakhutdinov, 2006

The setup shown in **Figure 6-5** is not drastically different from any of the feedforward neural networks we've used in the past. Let's start by describing the encoder half of the network. We first linearize the 28 by 28 image into a vector of size 784. The

encoder portion of the network has 3 internal layers, culminating in an embedding of size 2. We can use the same approach as we used to construct the hidden layers of a feedforward network. You'll notice that the code looks very similar to code we've written in the past, where x represents a minibatch of 784-dimensional input vectors.

```
def encoder(x, n_code, phase_train):
    with tf.variable_scope("encoder"):
        with tf.variable_scope("hidden_1"):
            hidden_1 = layer(x, [784, n_encoder_hidden_1],
                             [n_encoder_hidden_1], phase_train)

        with tf.variable_scope("hidden_2"):
            hidden_2 = layer(hidden_1, [n_encoder_hidden_1,
                                       n_encoder_hidden_2], [n_encoder_hidden_2], phase_train)

        with tf.variable_scope("hidden_3"):
            hidden_3 = layer(hidden_2, [n_encoder_hidden_2,
                                       n_encoder_hidden_3], [n_encoder_hidden_3], phase_train)

        with tf.variable_scope("code"):
            code = layer(hidden_3, [n_encoder_hidden_3, n_code],
                         [n_code], phase_train)

    return code
```

The decoder network is built with the same principle, but the 2-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 by 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 the previous chapter. 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 $\| I - O \| = \sqrt{\sum_i (I_i - O_i)^2}$, also known the L2 norm of the difference between the two vectors. We average this function over the whole minibatch 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 minibatch 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 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 2 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):
        minibatch_x, minibatch_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: minibatch_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
```

And navigate your browser to <http://localhost:6006/>. The results of “Graph” tab are shown in **Figure 6-6**.

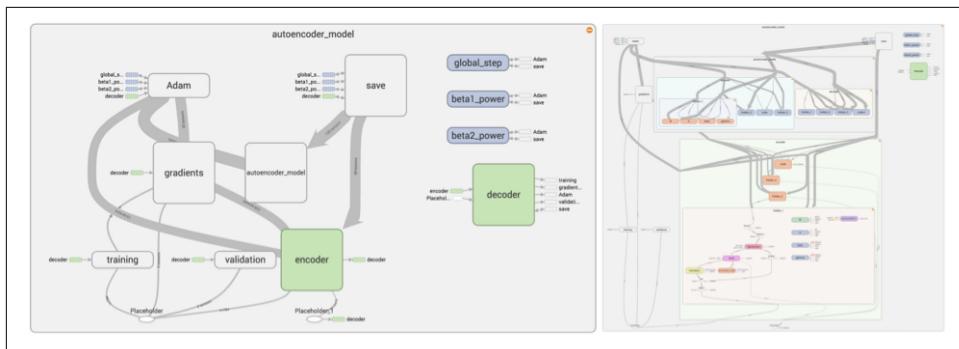


Figure 6-6. Tensorflow allows us to neatly view the high level components and data flow of our computation graph (left) and also click through to more closely inspect the data flows of individual subcomponents (right).

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 plateau'd at a "good" cost, or whether our model is still doing a poor job of reconstructing the original inputs

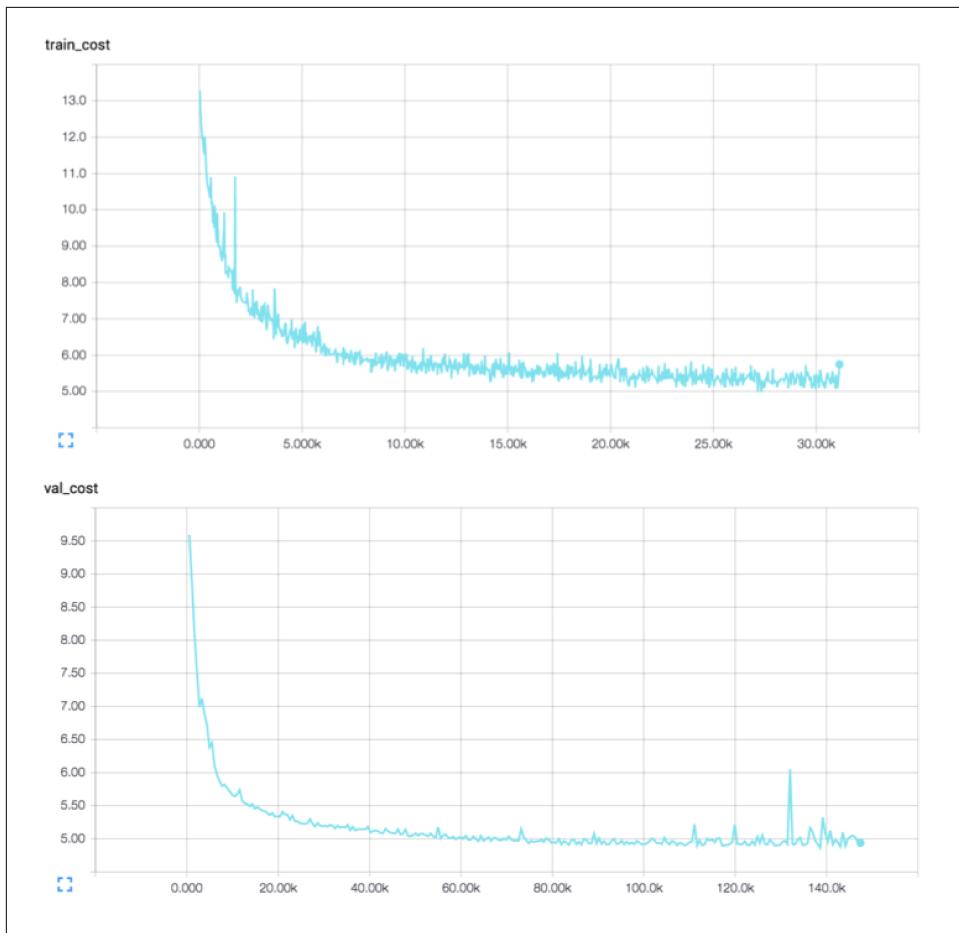


Figure 6-7. The cost incurred on the training set (logged after each minibatch) and on the validation set (logged after each epoch)

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.

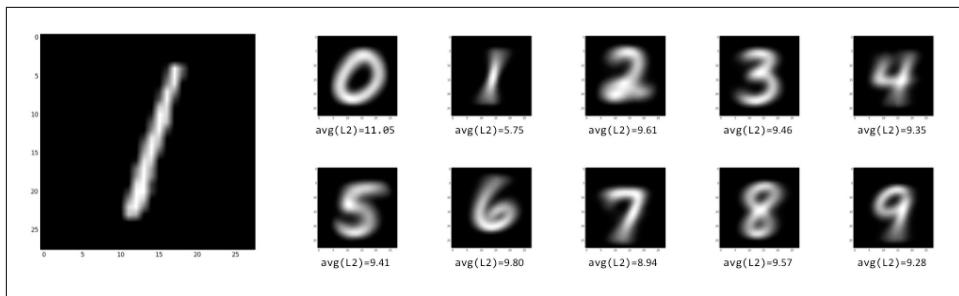


Figure 6-8. The image of the 1 on the left is compared to all of the other digits in the MNIST dataset. Each digit class is represented visually with the average of all of its members and labeled with the average of the L2 costs comparing the 1 on the left with all of the class's members.

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**.

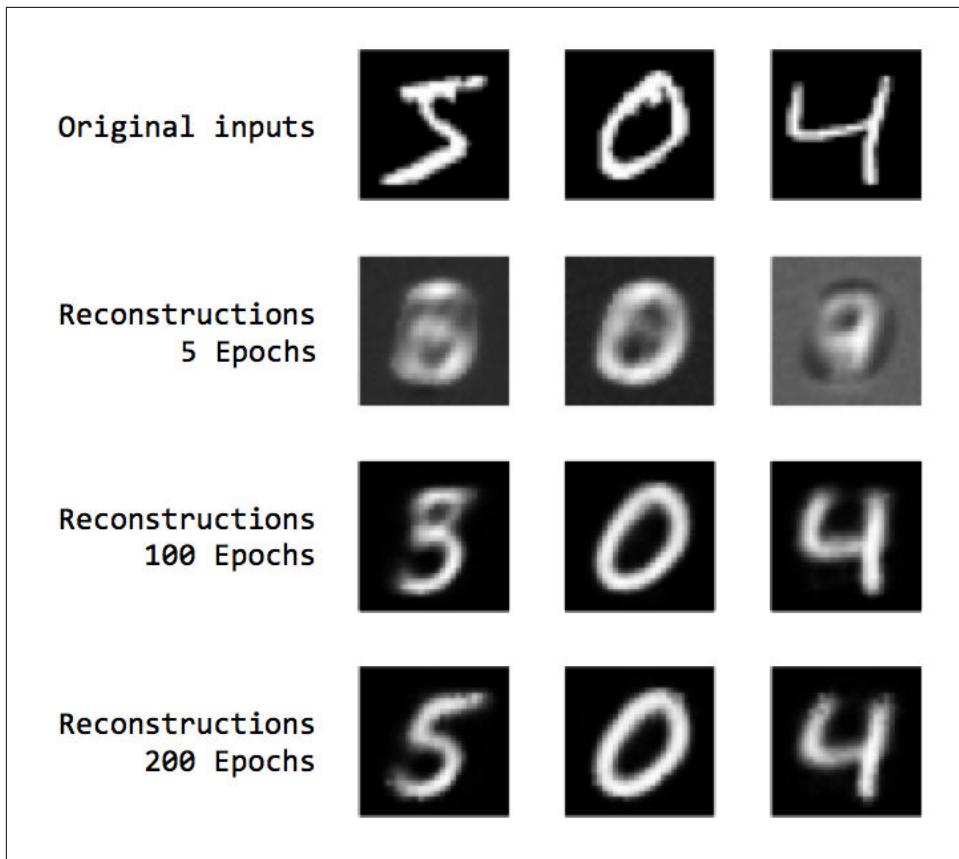


Figure 6-9. A side by side comparison of the original inputs (from the validation set) and reconstructions after 5, 100, and 200 epochs of training.

After 5 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 2-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 2-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 2-dimensional codes. We can also reconstruct the original images from the 2-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 above 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 2 dimensional codes. In fact the PCA's performance is somewhat reminiscent of the the autoencoder only 5 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.

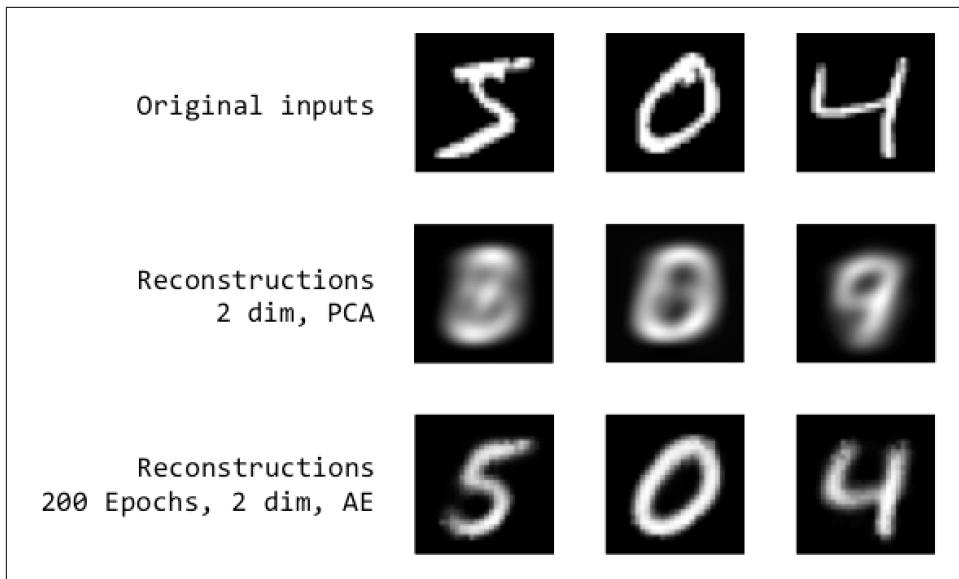


Figure 6-10. Comparing the reconstructions by both PCA and autoencoder side by side.

Now, to complete the experiment, we must load up a saved tensorflow model, retrieve the 2-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)

```

The resulting visualization in **Figure 6-11** is quite telling. Whereas it is extremely difficult to make out separable clusters in the 2-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 compared to PCA embeddings.

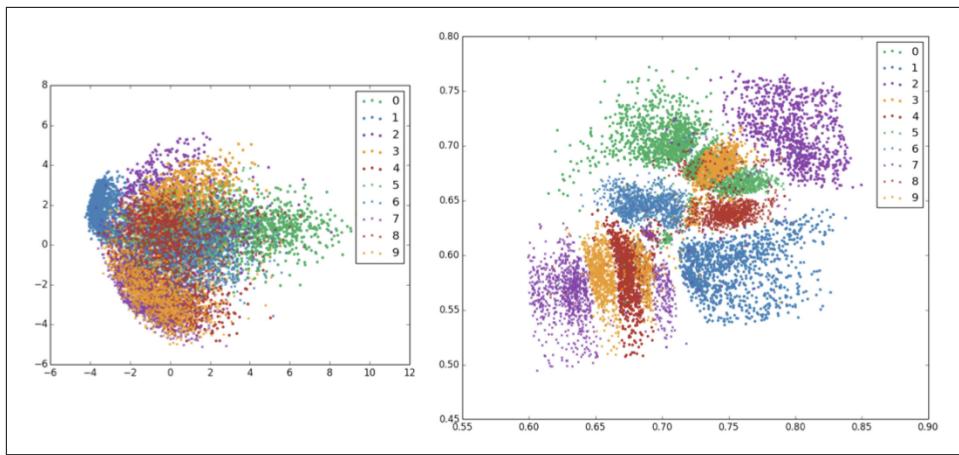


Figure 6-11. We visualize 2-dimensional embeddings produced by PCA (left) and by an autoencoder (right). Notice that the autoencoder does a much better job of clustering codes of different digit classes.

In this section, we successfully set up and trained a feedforward 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.

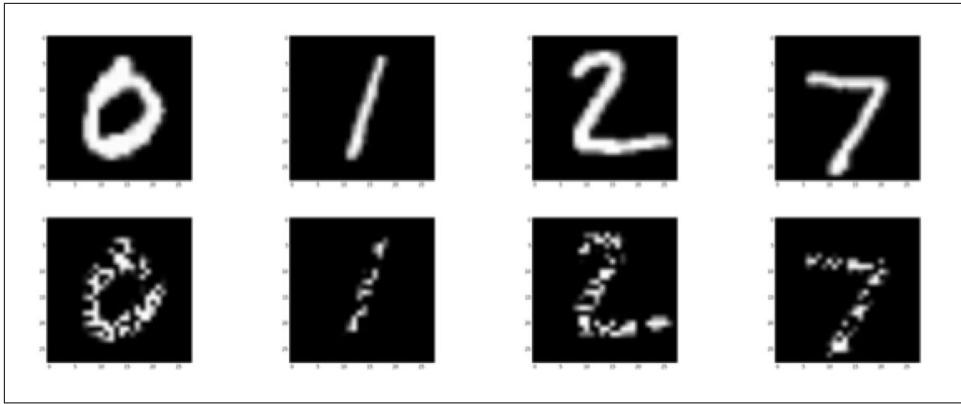


Figure 6-12. In the top row, we have original images from the MNIST dataset. In the bottom row, we've randomly blackened out half of the pixels. Despite the corruption, the digits in the bottom row are still identifiable by human perception.

One way to look at this phenomenon is probabilistic. 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 where they introduced the *denoising autoencoder*.

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 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 2-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 describe in 2014, our autoencoder is implicitly learning this manifold as it learns how to reconstruct data after pushing it through a bottle neck (the code layer). 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.

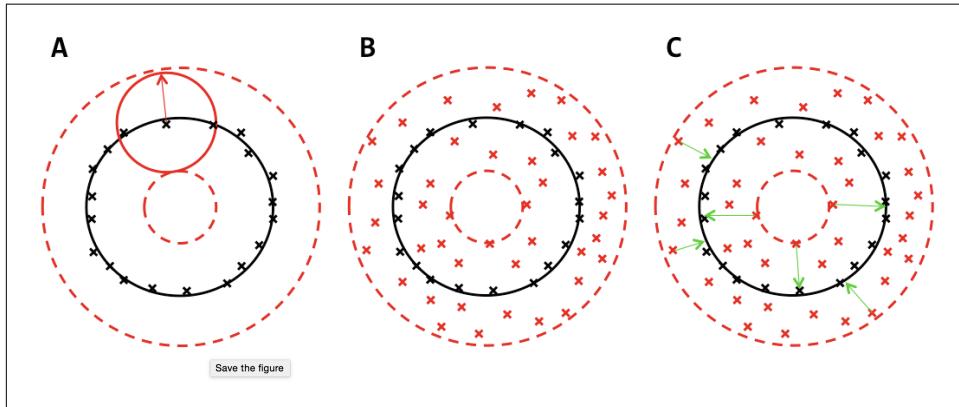


Figure 6-13. This is an image caption

As an illustration, let's consider the scenario in **Figure 6-13**, where the points in S 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 .

With the philosophical motivations of denoising in mind, we can now make a small modification 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 as we describe above if the `corrupt` placeholder is equal to 1 and refrain from corrupting the input if the `corrupt` placeholder tensor is equal to 0. After making this modification, we can re-run 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

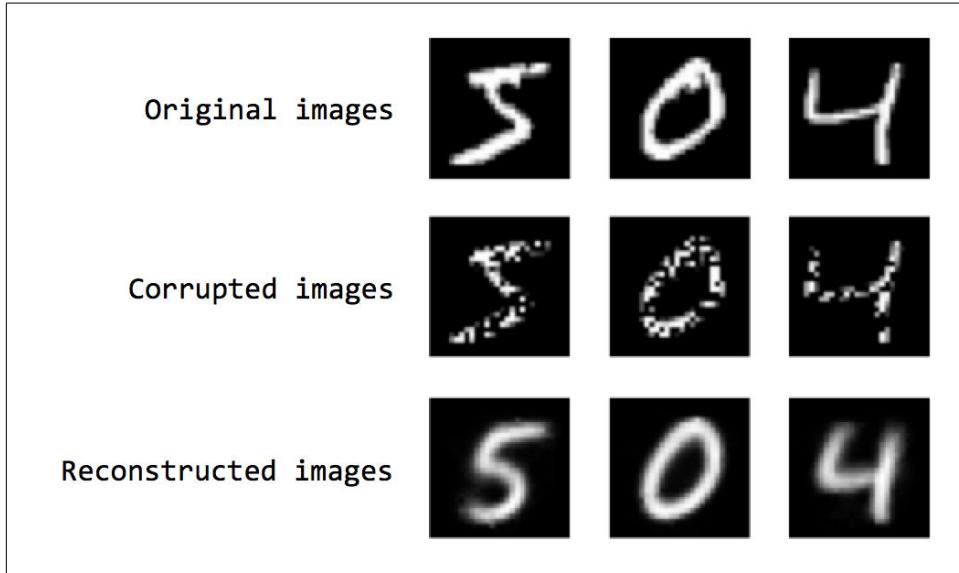


Figure 6-14. We apply a corruption operation to the dataset and train a denoising autoencoder to reconstruct the original, uncorrupted images.

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 a patient has or does not have cancer, the doctor will likely desire 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**.

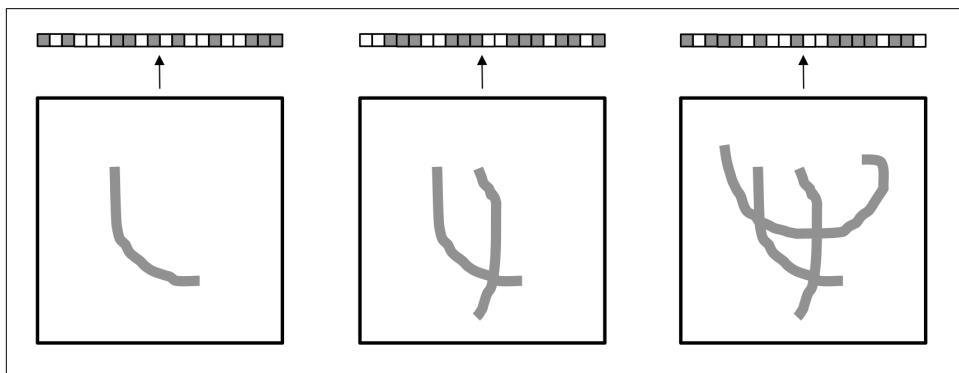


Figure 6-15. The activations of a dense representation combine and overlay information from multiple features in ways that are difficult to interpret.

The autoencoder produces a *dense* representation, i.e. 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 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.

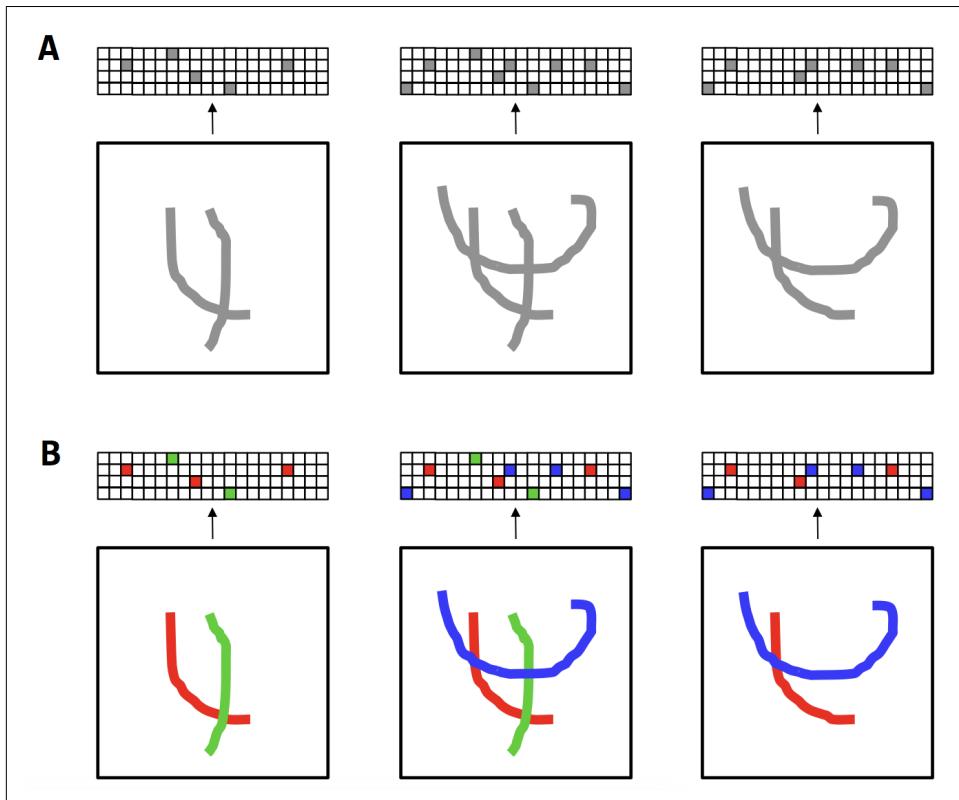


Figure 6-16. With the right combination of space and sparsity, a representation is more interpretable. In A, we show how activations in the representation change with the addition and removal of strokes. In B, we color code the activations that correspond to each stroke to highlight our ability to interpret how a stroke affects the representation.

While this is the ideal outcome, we'll have 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 the 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 achieve 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 strong we favor sparsity at the expense of generating better reconstructions. For the mathematically inclined, one 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 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 and 2008. More recently, a 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. 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 this 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 is extraneous to this fundamental understanding of the data.

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**.

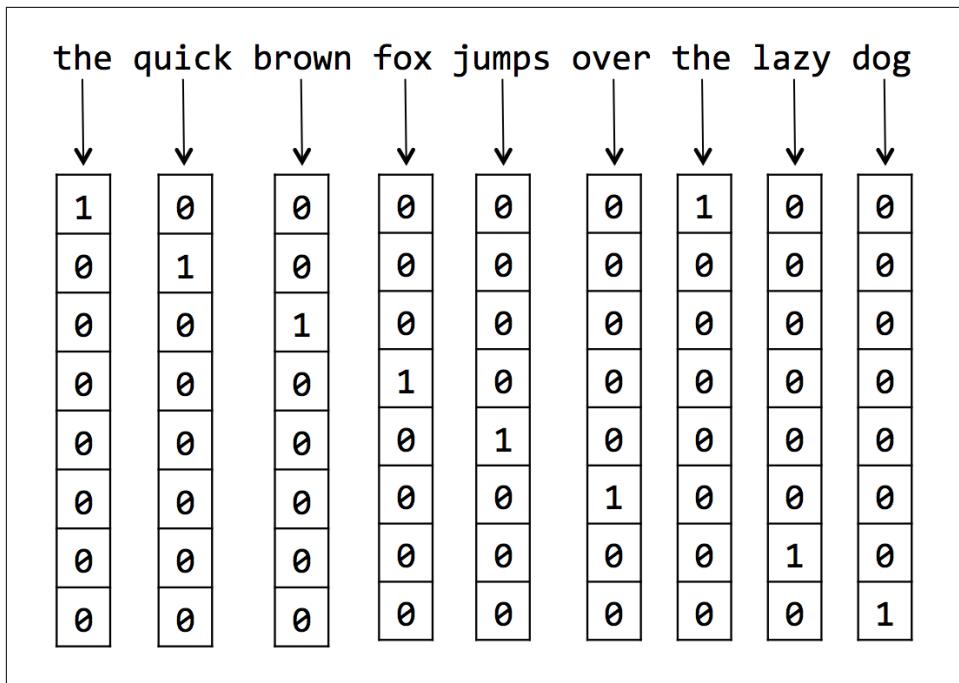


Figure 6-17. An example of generating one-hot vector representations for words using a simple document

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 1) discovering these relationships and 2) 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.

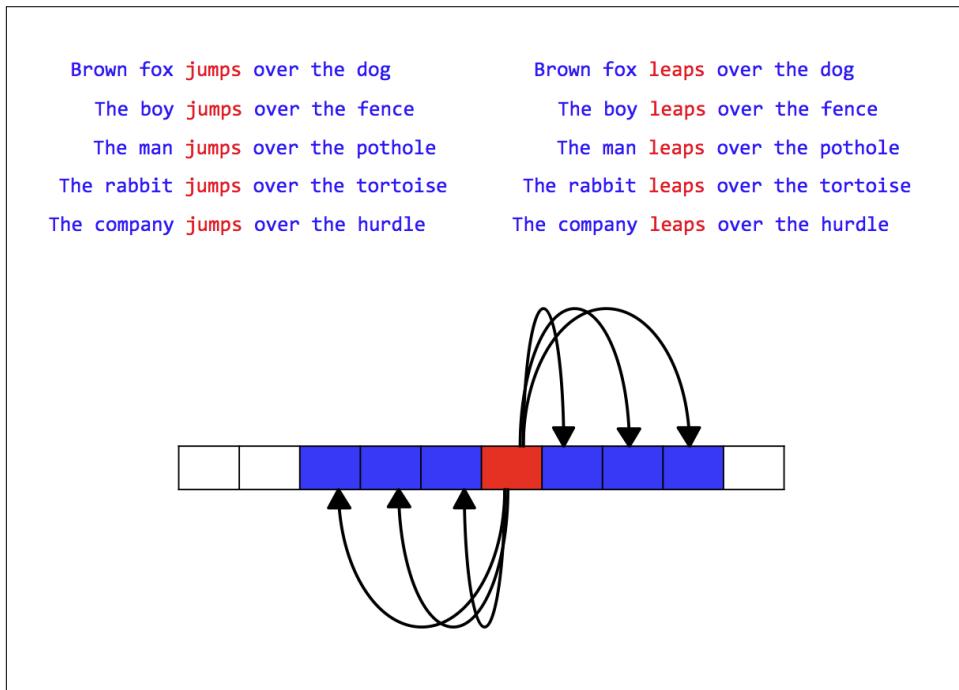


Figure 6-18. We can identify words with similar meanings based on their contexts. For example, the words “jumps” and “leaps” should have similar vector representations because they are virtually interchangeable. Moreover, we can draw conclusions over what the words “jumps” and “leaps” mean just by looking 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.

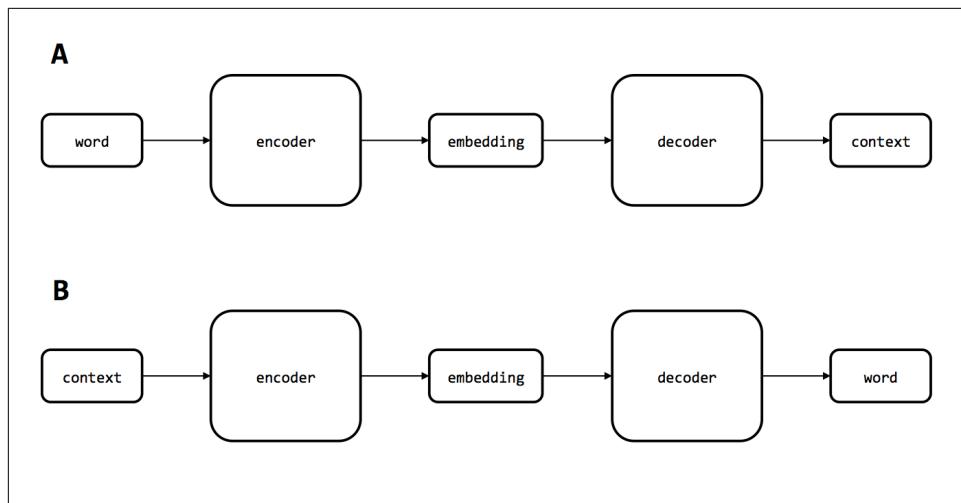


Figure 6-19. General architectures for designing encoders and decoders that generate embeddings by mapping words to their respective contexts (A) or vice versa (B)

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. This model is much like strategy B from the previous section. The CBOW model using 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. The Skip-Gram model does the inverse of CBOW, taking the target word as an 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 $i \in \{0, 1, \dots, |V| - 1\}$ 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 look up 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.

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 feedforward 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.

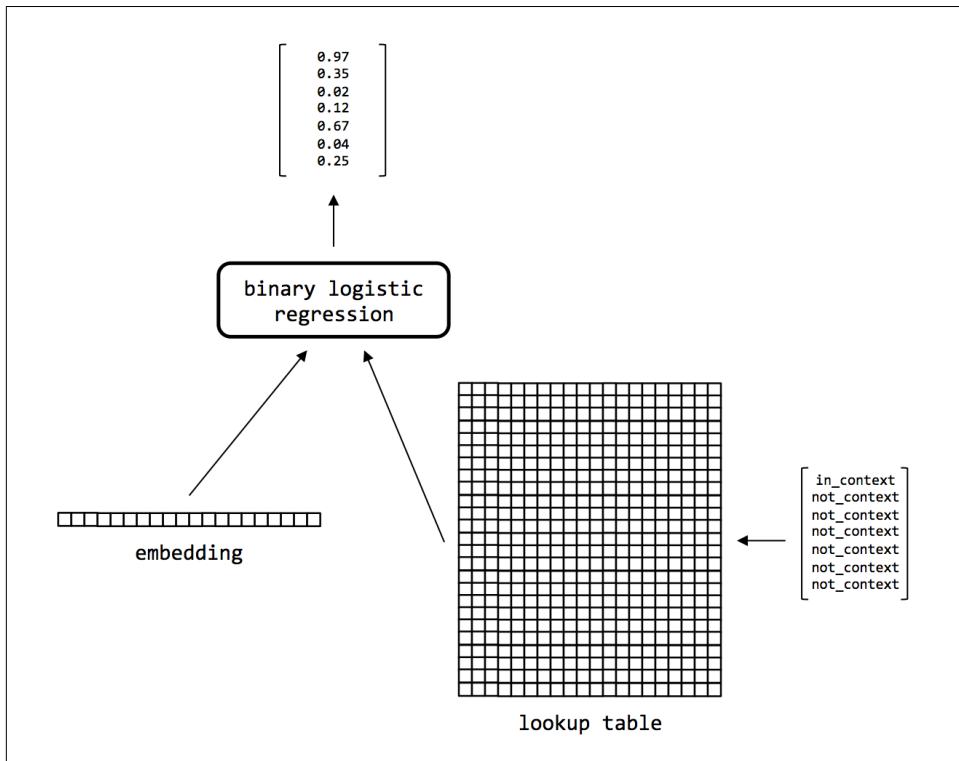


Figure 6-20. An illustration of how noise-contrastive estimation works. A binary logistic regression compares the embedding of the target with the embedding of a context word and randomly sampled non-context words. We construct a loss function describing how effectively the embeddings enable identification of words in the context of the target vs. words outside the context of the target.

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**. 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 non-context 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 next chapter, 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 next chapter, 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 have use a

context window of 5 words to the left and to the right of each target word, and sample 4 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, 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 construct 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 above 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

        minibatch_x, minibatch_y = data.generate_batch(batch_size,
                                                       num_skips, skip_window)
        feed_dict = {x : minibatch_x, y : minibatch_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 start 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 complete, 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.”

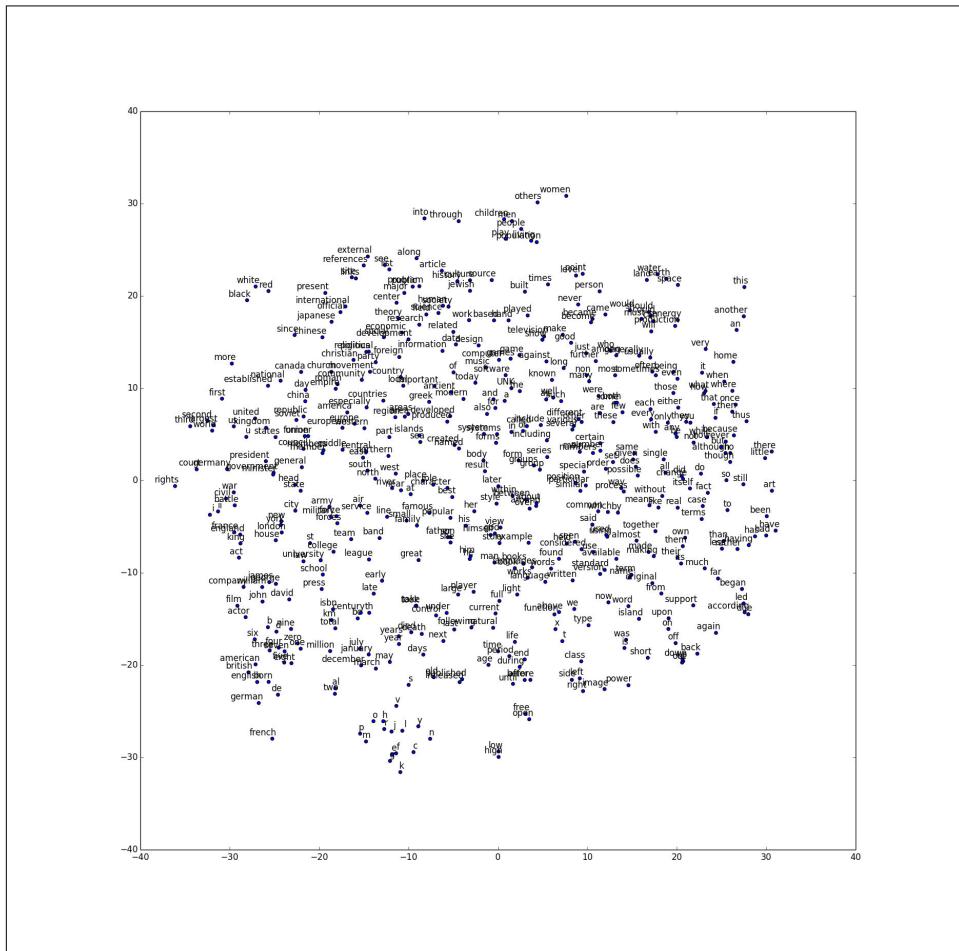


Figure 6-21. Visualization of our Skip-Gram embeddings using t-SNE. We notice that similar concepts are closer together than disparate concepts, indicating that our embeddings encode meaningful information about the functions and definitions of individual words

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 the previous chapter 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.asarray(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.

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.

