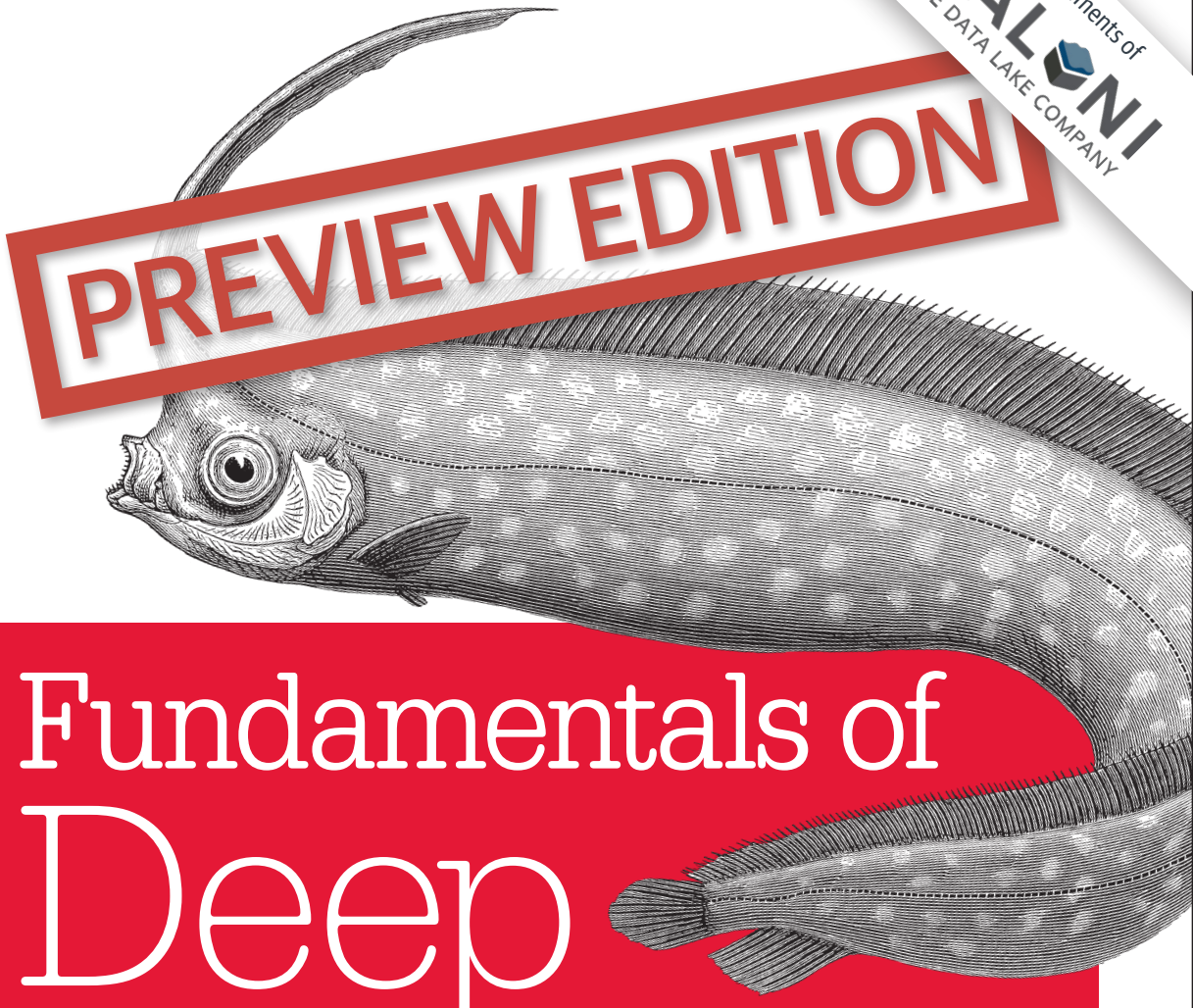
O'REILLY®

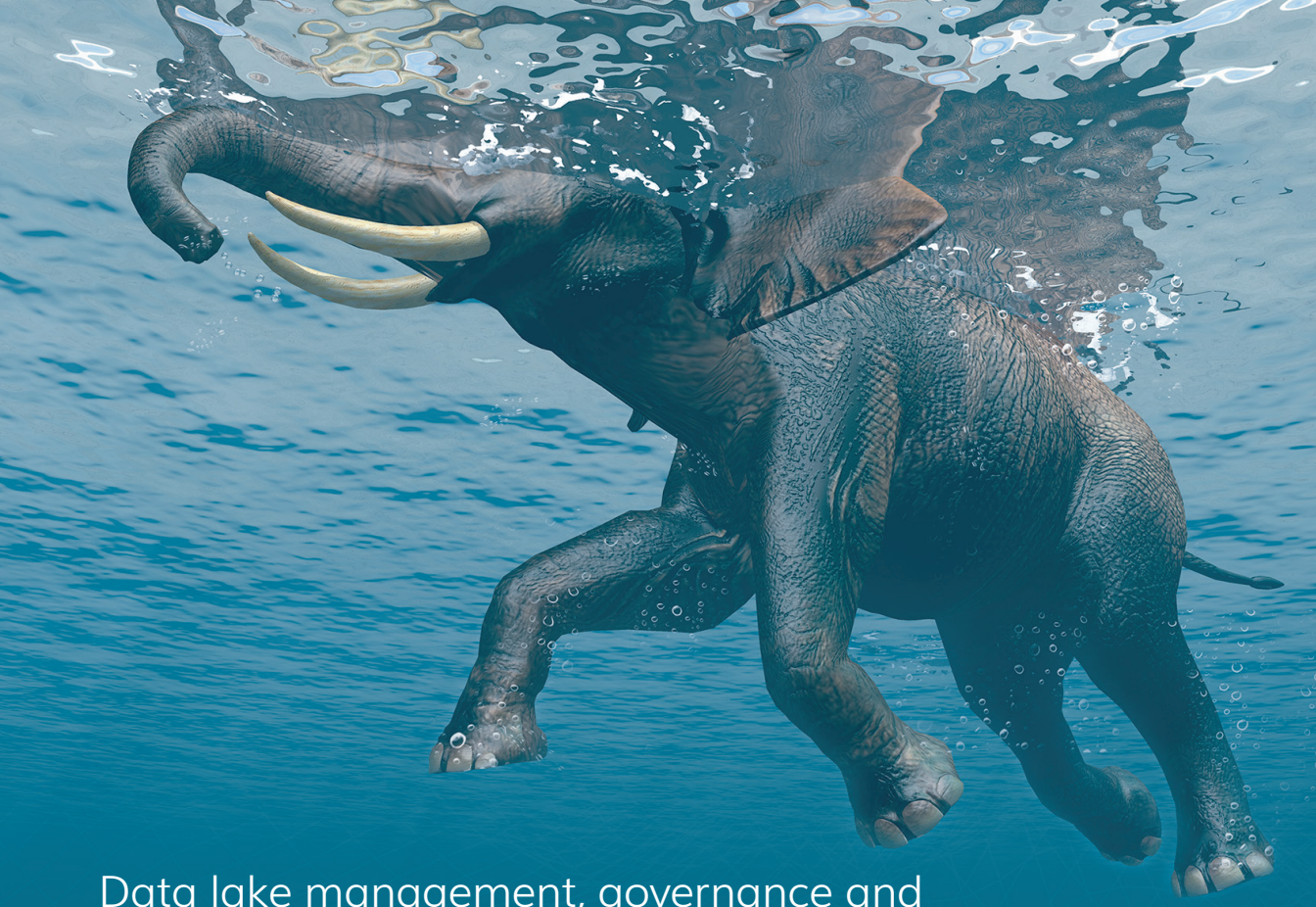
PREVIEW EDITION



Fundamentals of Deep Learning

DESIGNING NEXT-GENERATION
ARTIFICIAL INTELLIGENCE ALGORITHMS

Nikhil Buduma



Data lake management, governance and self-service data preparation

DON'T GO IN THE LAKE WITHOUT US

Learn More

www.zaloni.com

ZALONI

Fundamentals of Deep Learning

Designing Next Generation Artificial Intelligence Algorithms

This Preview Edition of *Fundamentals of Deep Learning*, Chapters 1–3, 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

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:
Copyeditor:
Proofreader:

Indexer:
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Panzer

November 2015: First Edition

Revision History for the First Edition
2015-06-12 First Early Release
2015-07-23 Second Early Release

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

978-1-491-92561-4

[LSI]

Table of Contents

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.

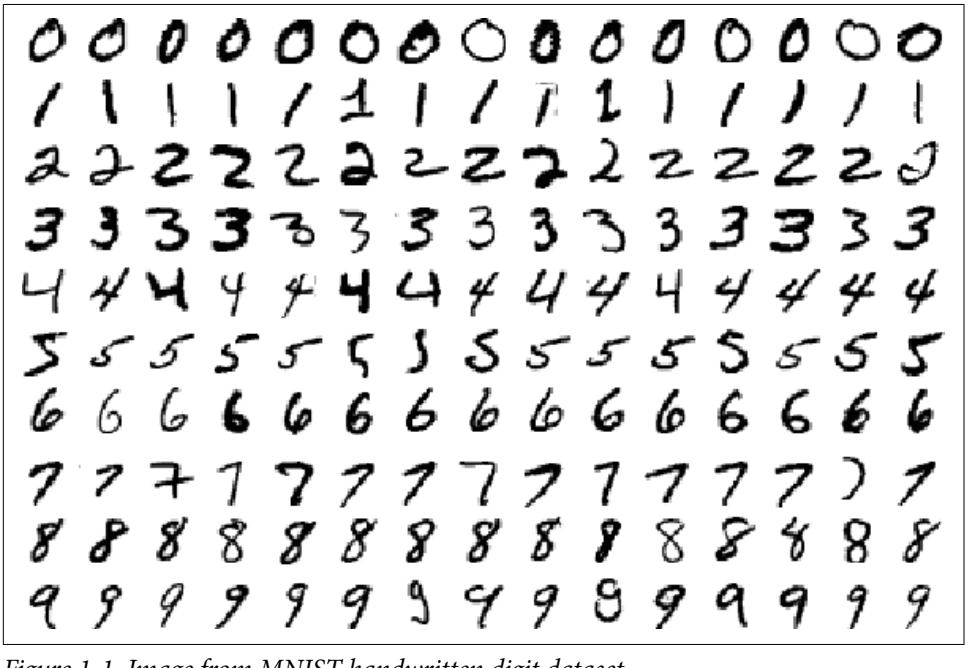


Figure 1-1. Image from MNIST handwritten digit dataset

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?

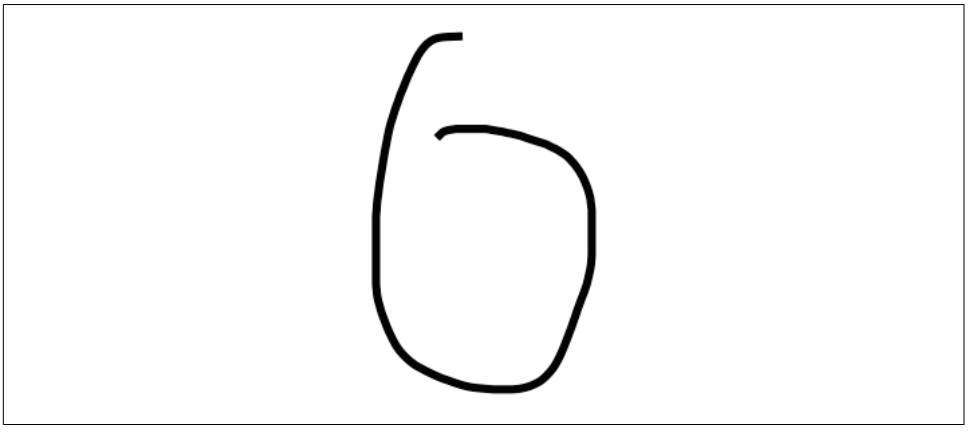


Figure 1-2. A zero that's difficult to distinguish from a six algorithmically

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**.

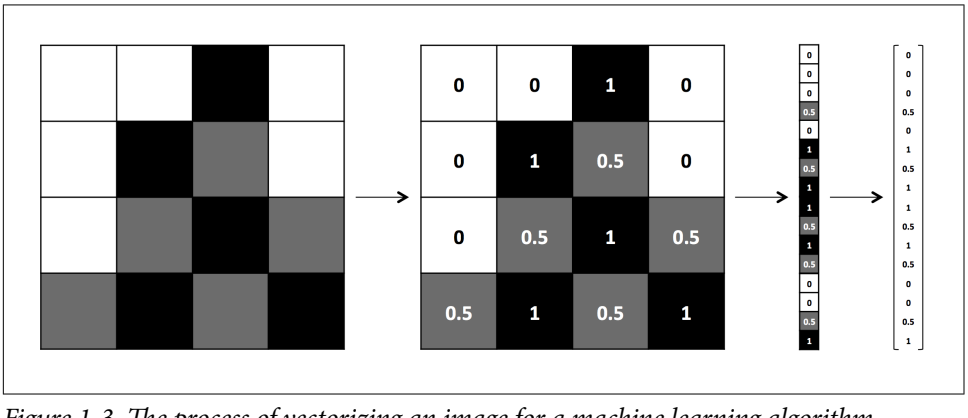


Figure 1-3. The process of vectorizing an image for a machine learning algorithm

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 $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$, we record the number of hours of sleep we got (x_1), the number of hours we spent studying (x_2), and whether we performed above average or below the class average. Our goal, then, might be to learn a model $h(\mathbf{x}, \theta)$ with parameter vector $\theta = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 \end{bmatrix}^T$ such that:

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

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

Then it turns out, by selecting $\theta = \begin{bmatrix} -24 & 3 & 4 \end{bmatrix}^T$, 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.

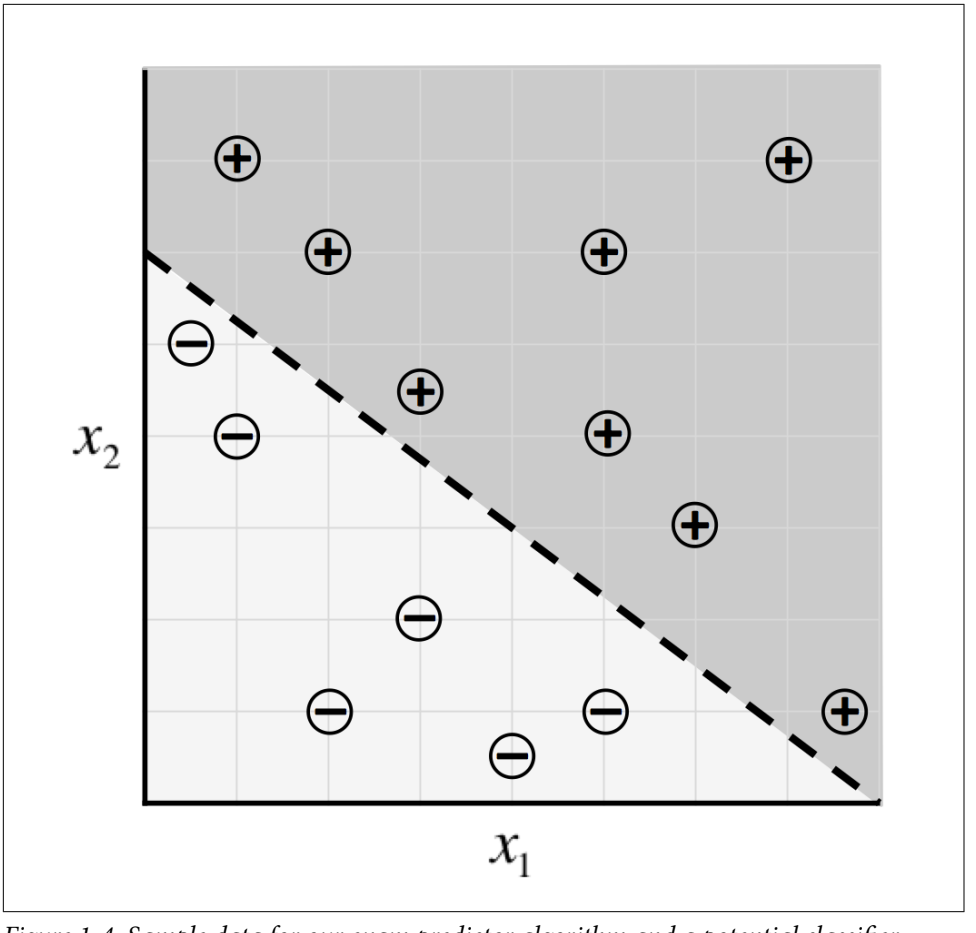


Figure 1-4. Sample data for our exam predictor algorithm and a potential classifier

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.

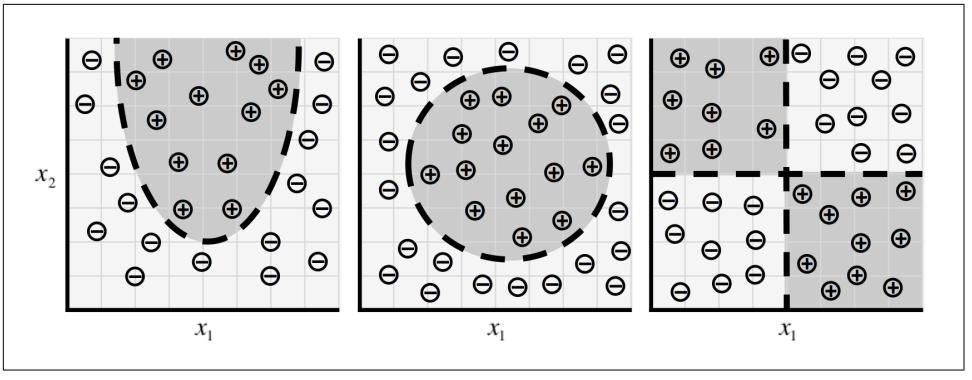


Figure 1-5. As our data takes on more complex forms, we need more complex models to describe them

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 of the brain, about the size of grain of rice, contains over 10,000 neurons, each of which forms an average of 6,000 connections with other neurons. 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.

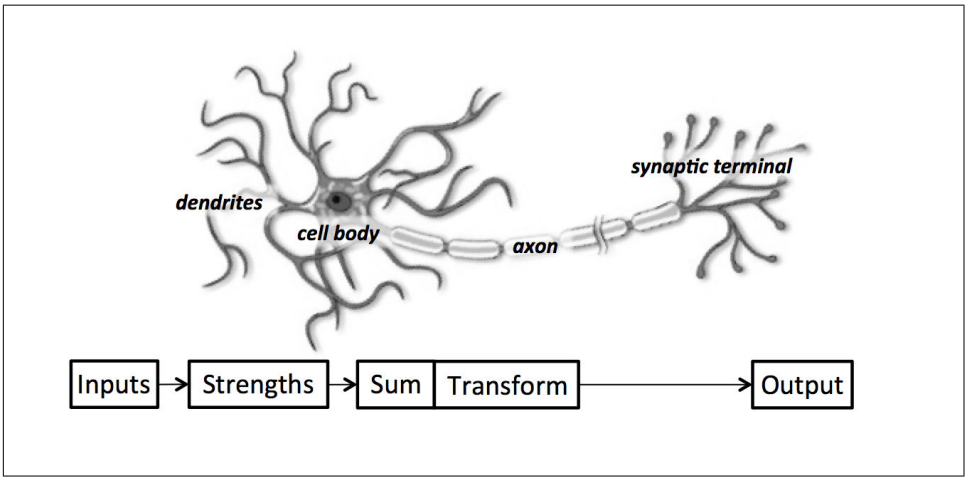


Figure 1-6. A functional description of a biological neuron's structure

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, $x_1, x_2, \ldots, x_n$, each of which is multiplied by a specific weight, $w_1, w_2, \ldots, w_n$. These weighted inputs are, as before, summed together to produce the *logit* of the neuron, $z = \sum_{i=0}^{n} w_i x_i$. In many cases, the logit also includes a *bias*, which is a constant (not shown in 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.

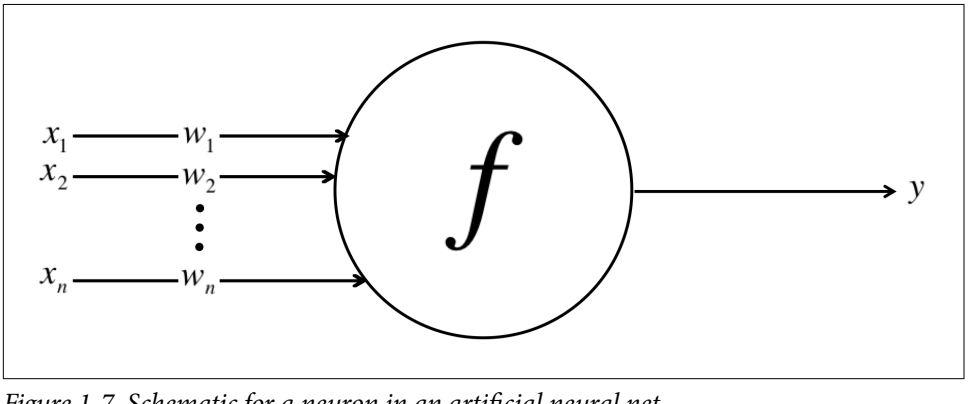


Figure 1-7. Schematic for a neuron in an artificial neural net

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.

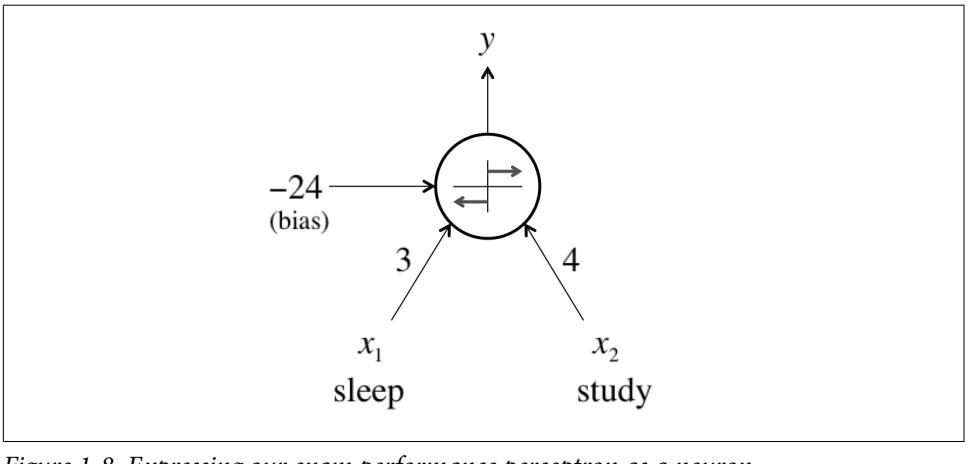


Figure 1-8. Expressing our exam performance perceptron as a neuron

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.

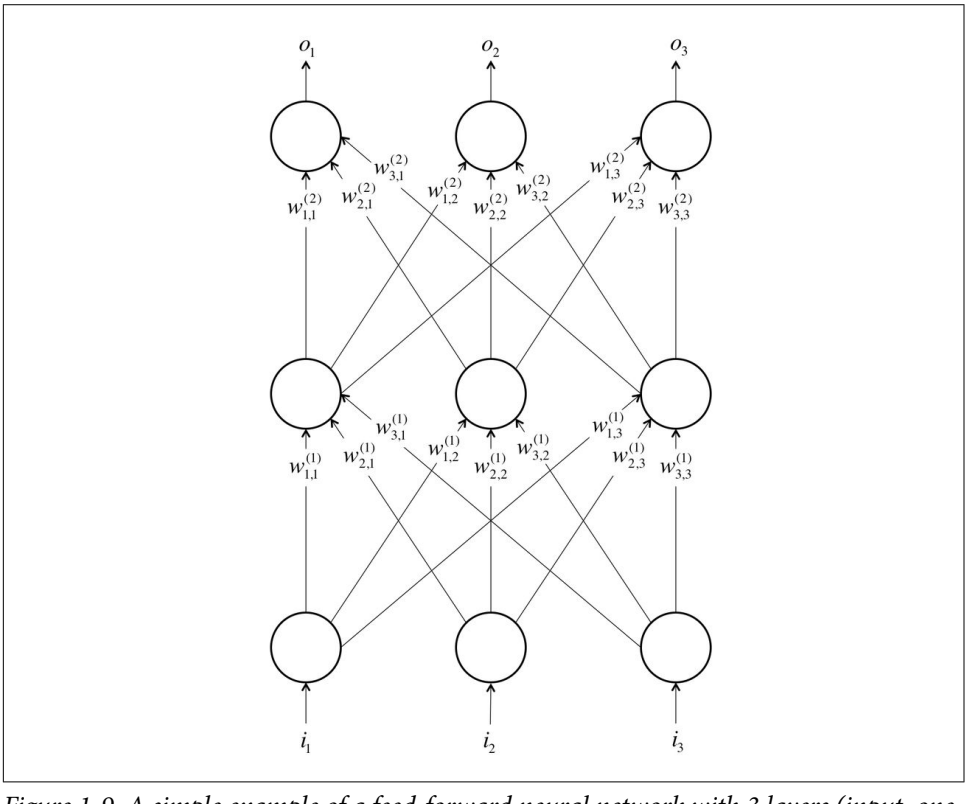


Figure 1-9. A simple example of a feed-forward neural network with 3 layers (input, one hidden, and output) and 3 neurons per layer

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 $w^{(k)}_{i,j}$ 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.

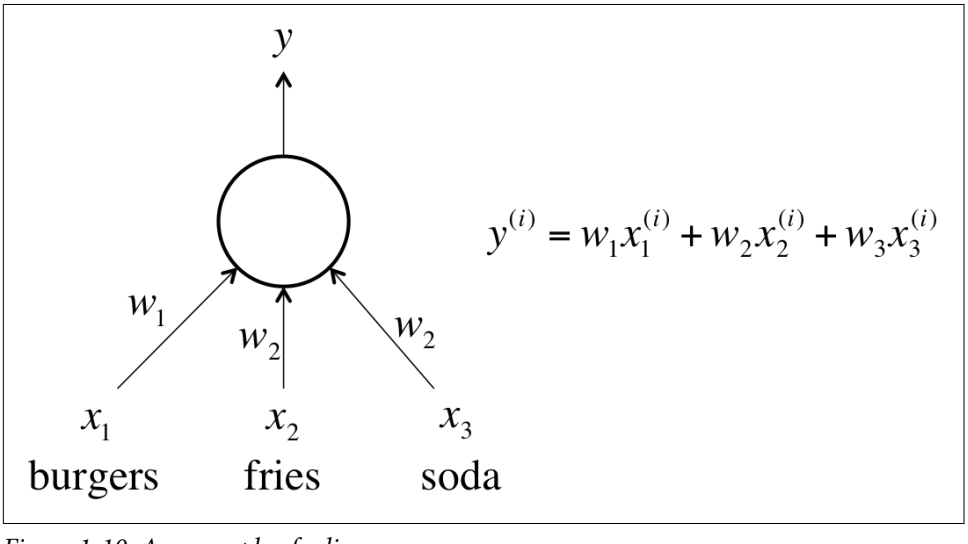


Figure 1-10. An example of a linear neuron

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:

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

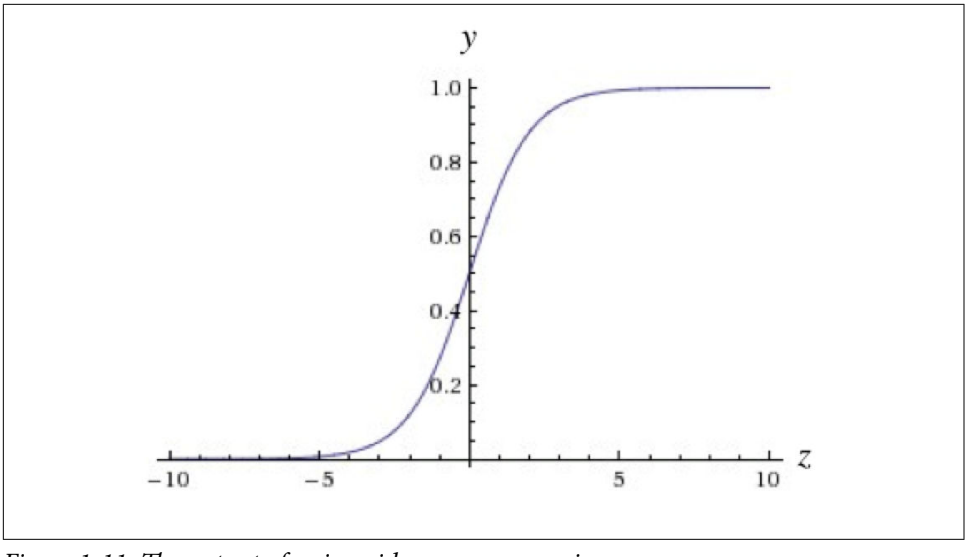


Figure 1-11. The output of a sigmoid neuron as z varies

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**.

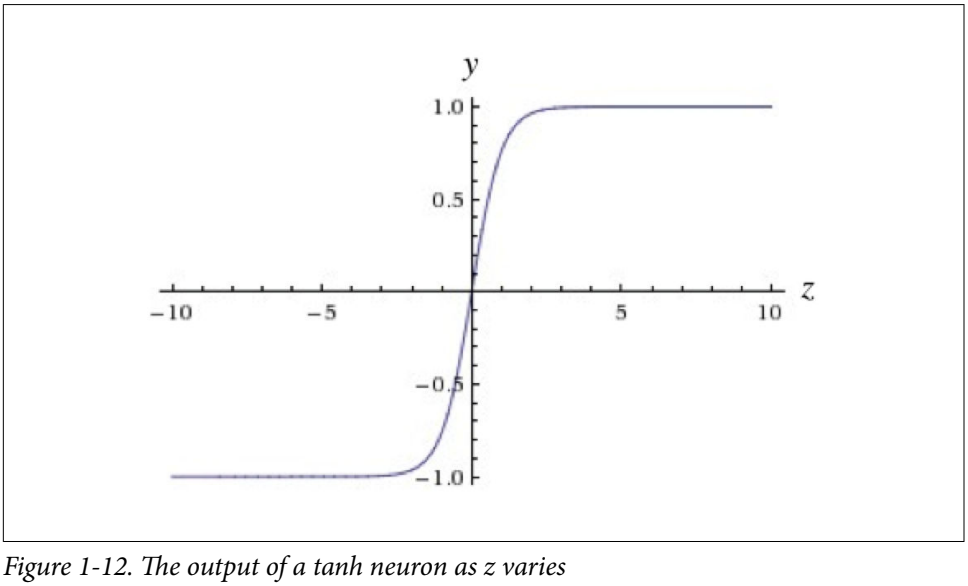


Figure 1-12. The output of a tanh neuron as z varies

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**.

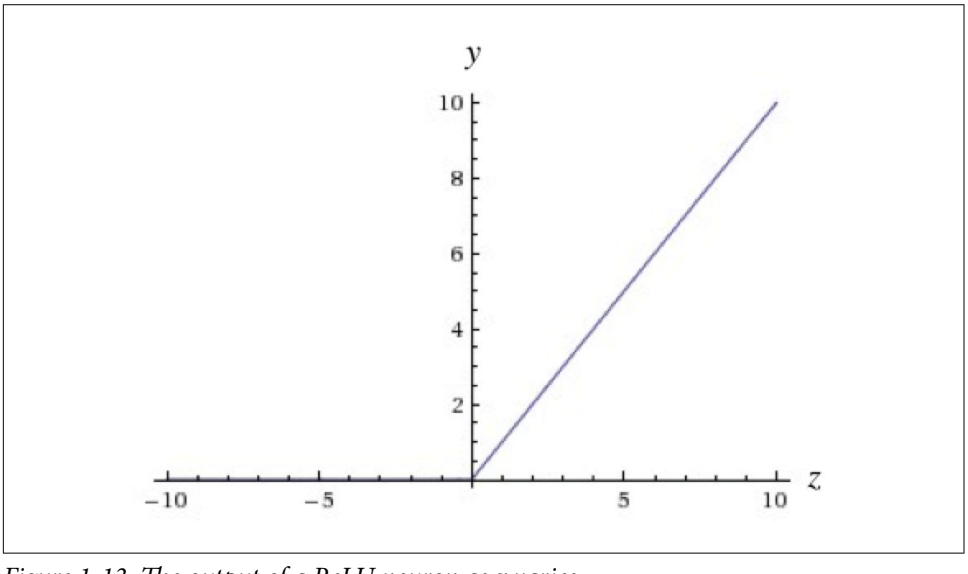


Figure 1-13. The output of a ReLU neuron as z varies

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 $\sum_{i=0}^{9} p_i = 1$:

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

This is achieved by using a special output layer called a softmax layer. Unlike in other kinds of layers, the output of a neuron in a softmax layer depends on the outputs of all 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:

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

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

Looking Forward

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

CHAPTER 2

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.

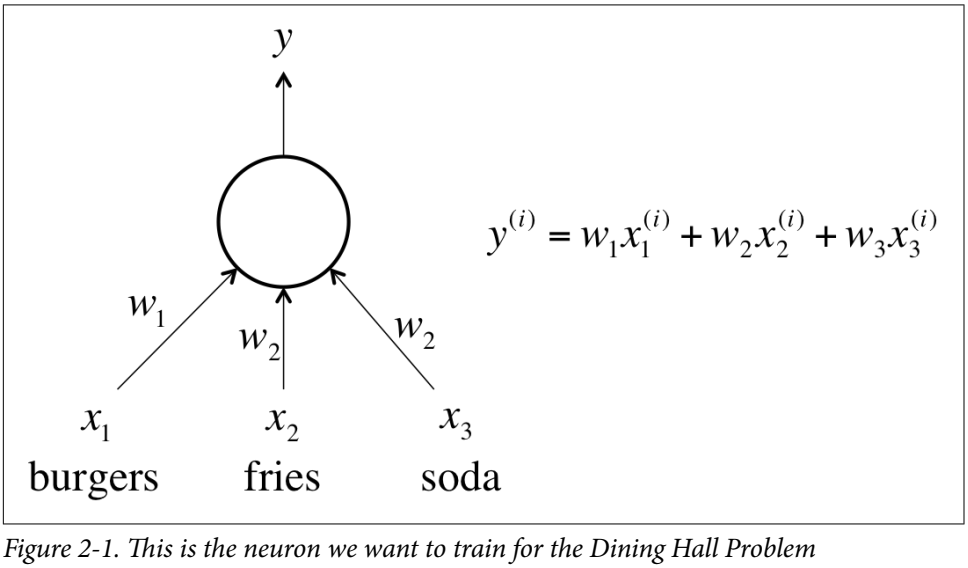


Figure 2-1. This is the neuron we want to train for the Dining Hall Problem

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:

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

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

Now at this point you might be wondering why we need to bother ourselves with error functions when we can treat this problem as a system of equations. After all, we have a bunch of unknowns (weights) and we have a set of equations (one for each

training example). That would automatically give us an error of 0 assuming that we have a consistent set of training 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**.

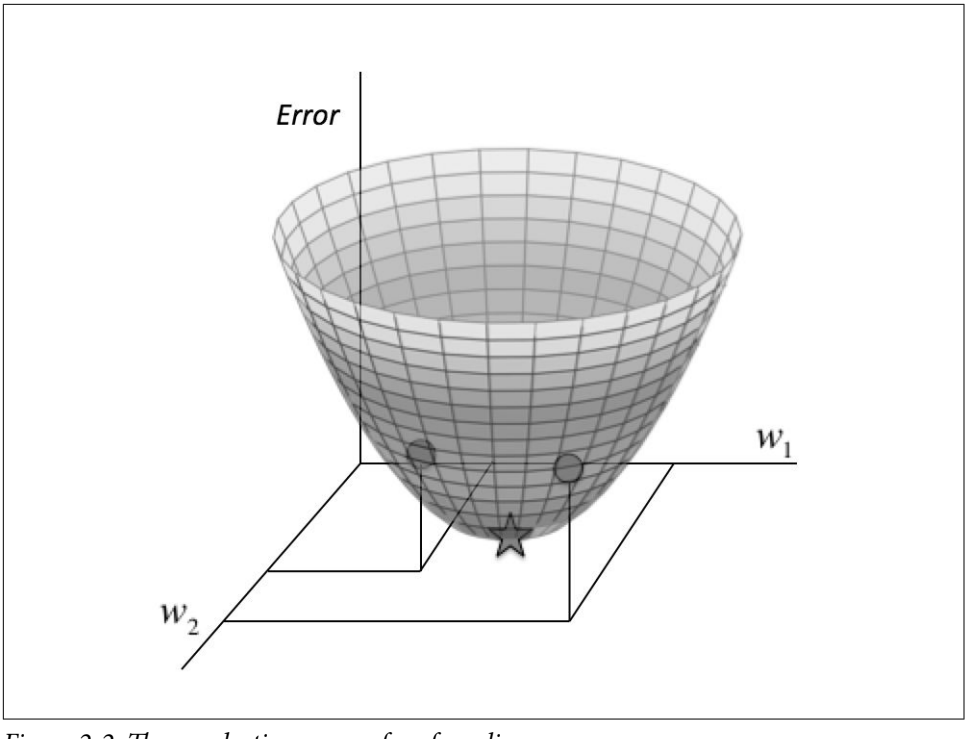


Figure 2-2. The quadratic error surface for a linear neuron

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*.

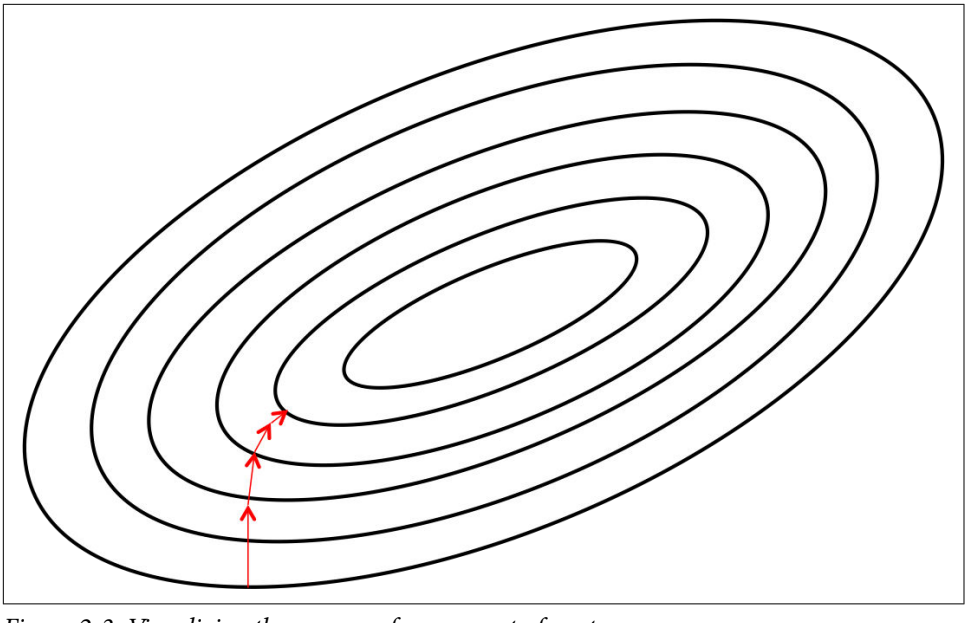


Figure 2-3. Visualizing the error surface as a set of contours

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.

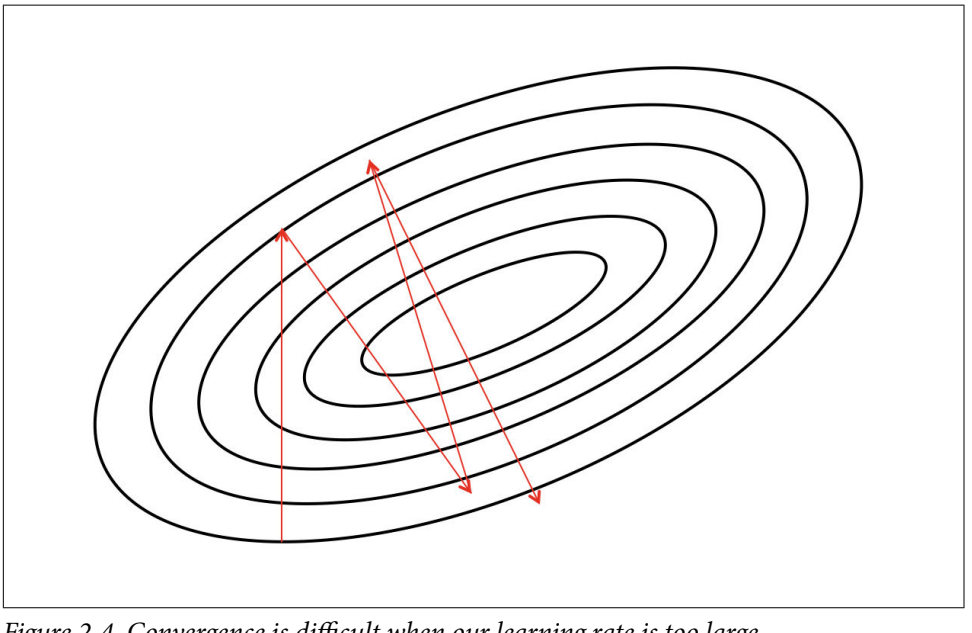


Figure 2-4. Convergence is difficult when our learning rate is too large

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:

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

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

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

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

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

Gradient Descent with Sigmoidal Neurons

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

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

$$z = \Sigma_k w_k x_k$$

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

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

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

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

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

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

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

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

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

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

Thus, the final rule for modifying the weights becomes:

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

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

The Backpropagation Algorithm

Now we're finally ready to tackle the problem of training multilayer neural networks (instead of just single neurons). 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 activites 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:

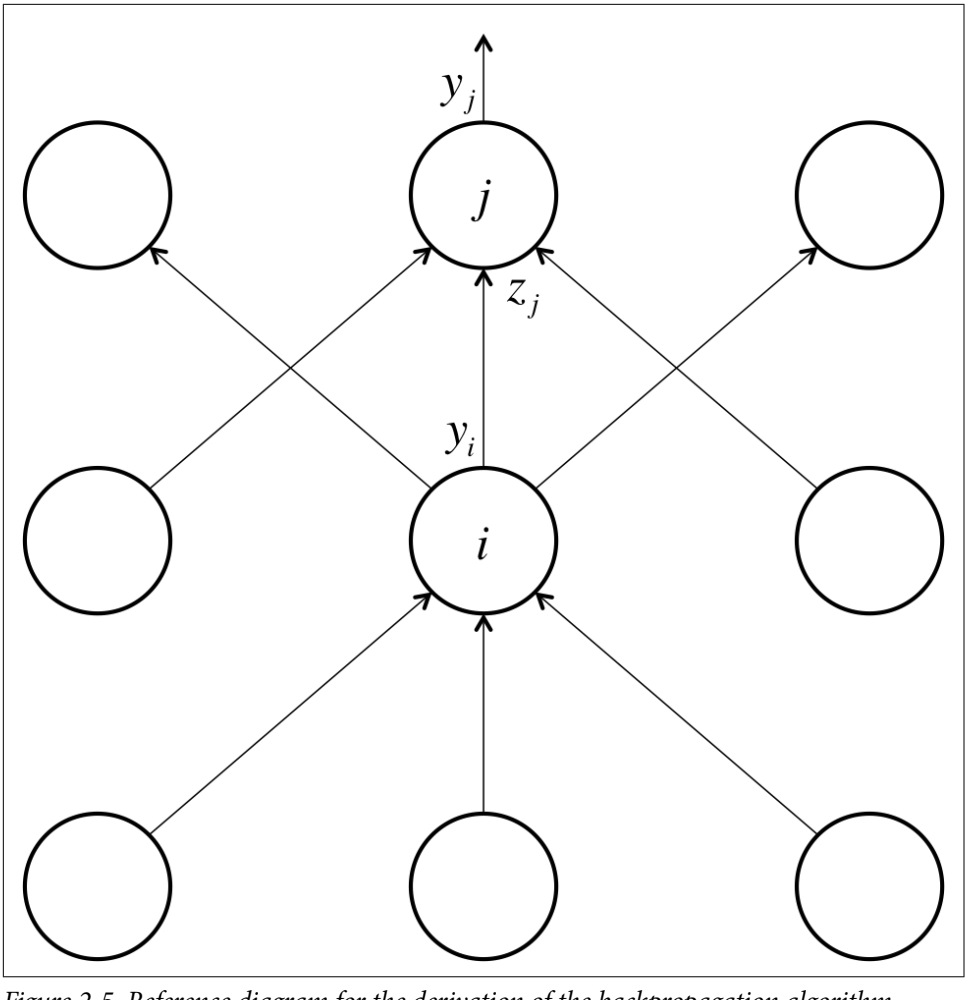


Figure 2-5. Reference diagram for the derivation of the backpropagation algorithm

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:

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

Now we tackle the inductive step. Let's presume we have the error derivatives for layer *j*. We now aim to calculate the error derivatives for the layer below it, layer *i*. To do so, we must accumulate information about how the output of a neuron in layer *i* affects the logits of every neuron in layer *j*. This can be done as follows, using the fact that the partial derivative of the logit with respect to the incoming output data from the layer beneath is merely the weight of the connection w_{ij}:

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

Furthermore, we observe the following:

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

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

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

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

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

Finally to complete the algorithm, 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:

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

This completes our description of the backpropagation algorithm!

Stochastic and 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.

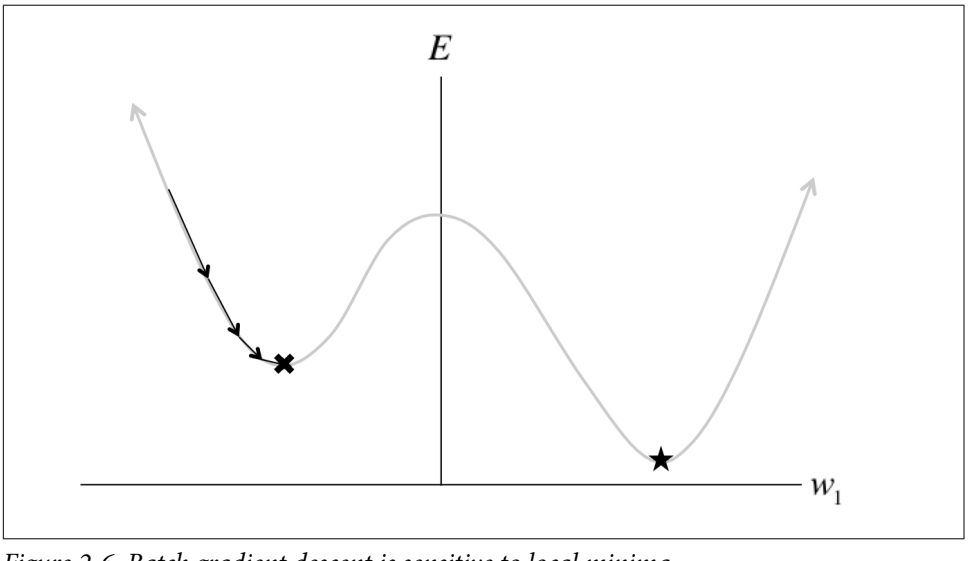


Figure 2-6. Batch gradient descent is sensitive to local minima

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.

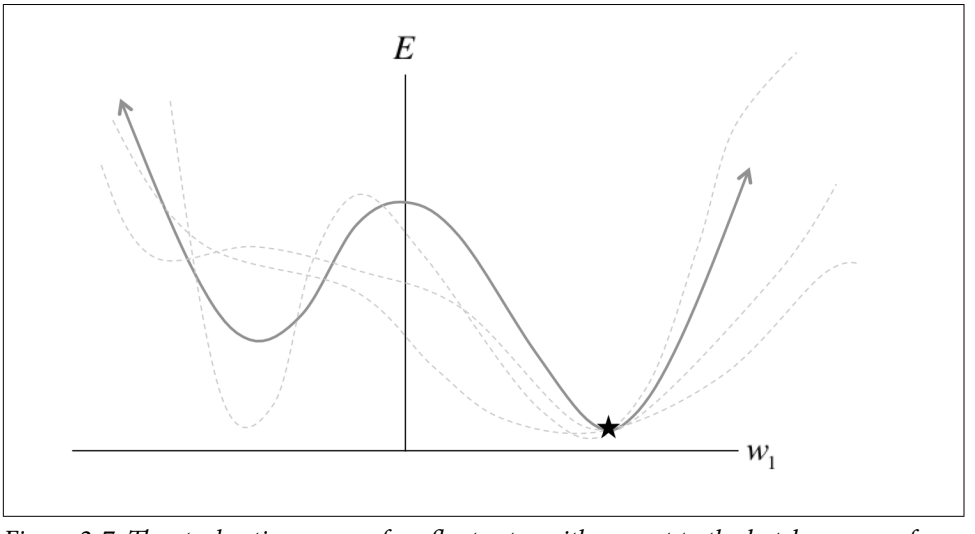


Figure 2-7. The stochastic error surface fluctuates with respect to the batch error surface, enabling local-minima avoidance

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:

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

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**.

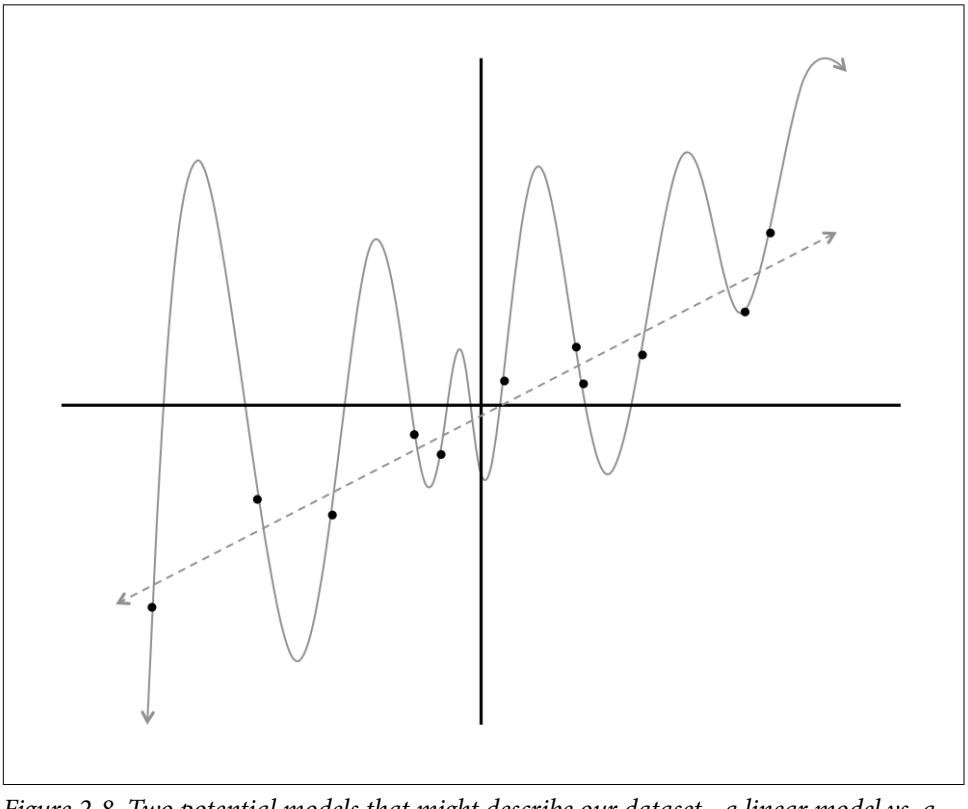


Figure 2-8. Two potential models that might describe our dataset - a linear model vs. a degree 12 polynomial

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.**

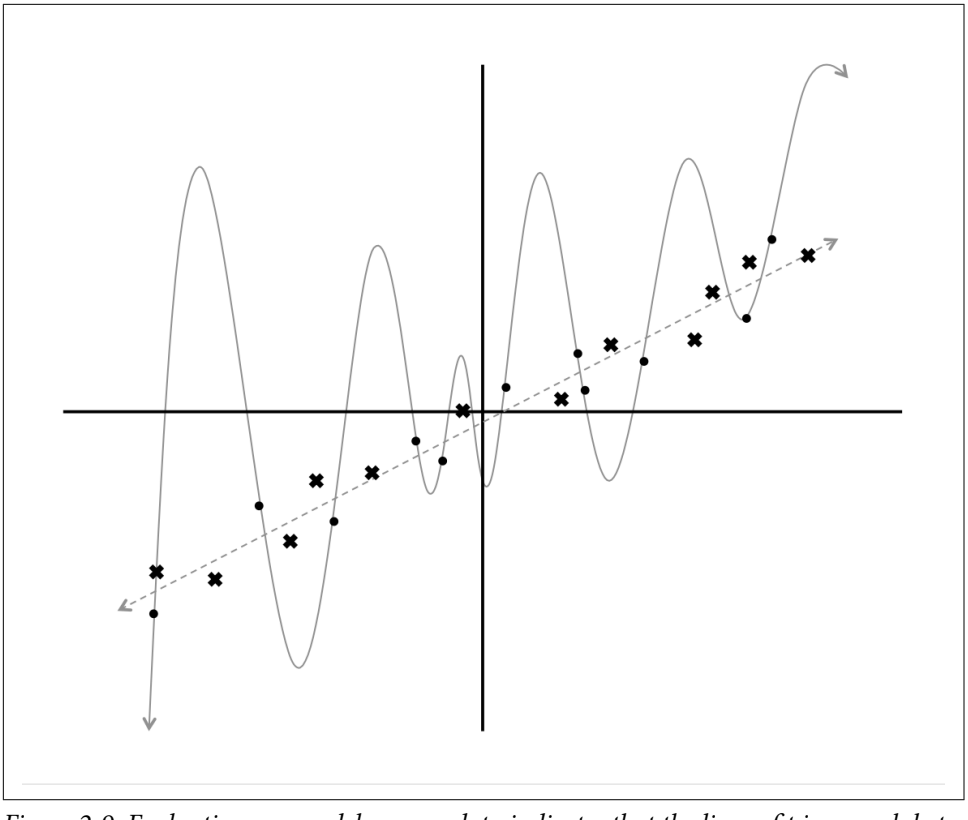


Figure 2-9. Evaluating our model on new data indicates that the linear fit is a much better model than the degree 12 polynomial

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 show in **Figure 2-10**.

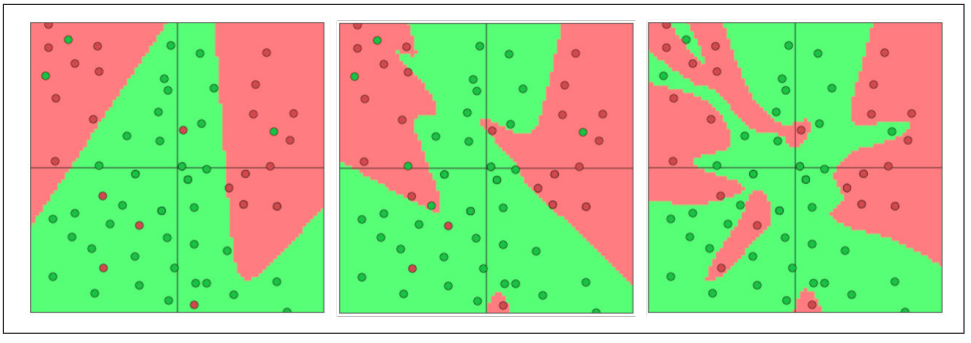


Figure 2-10. A visualization of neural networks with 3, 6, and 20 neurons (in that order) in their hidden layer.

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.

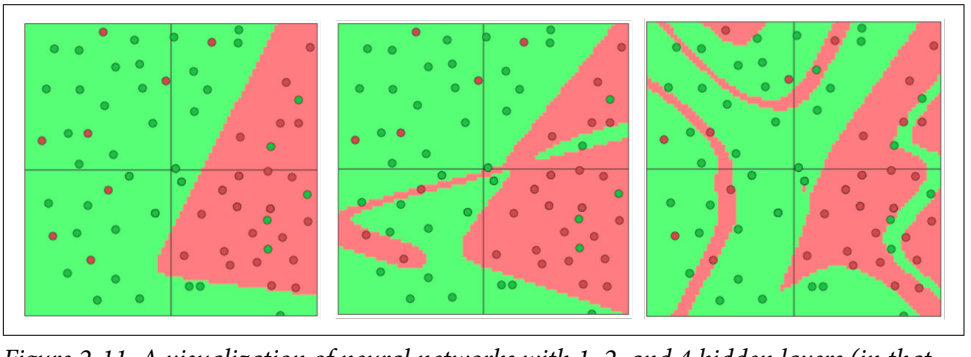


Figure 2-11. A visualization of neural networks with 1, 2, and 4 hidden layers (in that order) of 3 neurons each.

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*.

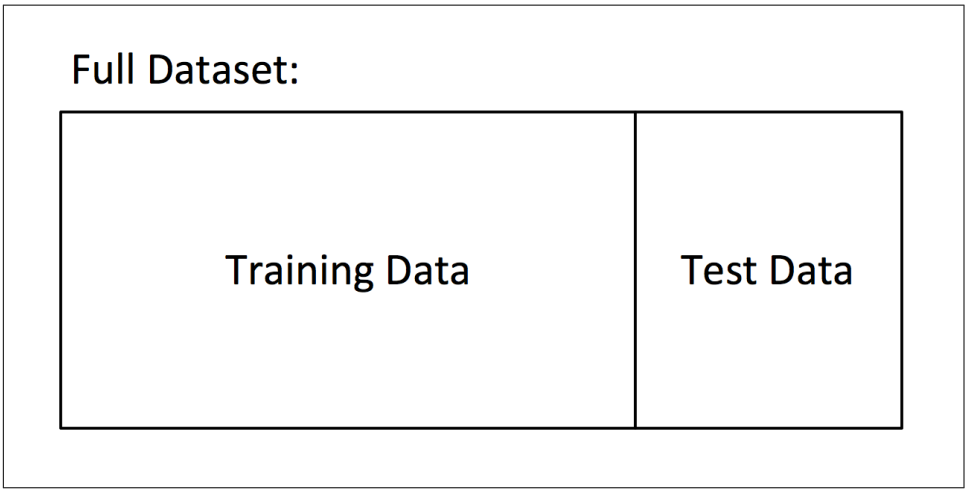


Figure 2-12. We often split our data into non-overlapping training and test sets in order to fairly evaluate our model

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.

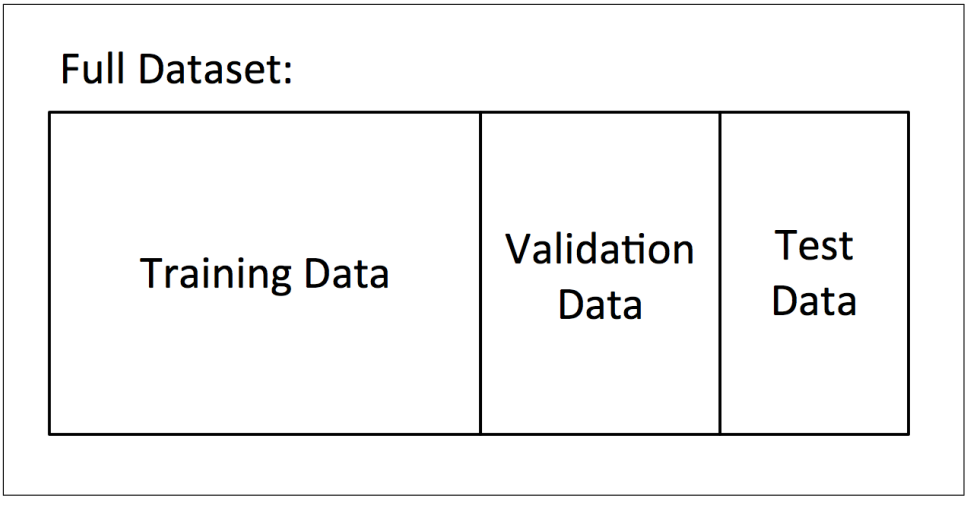


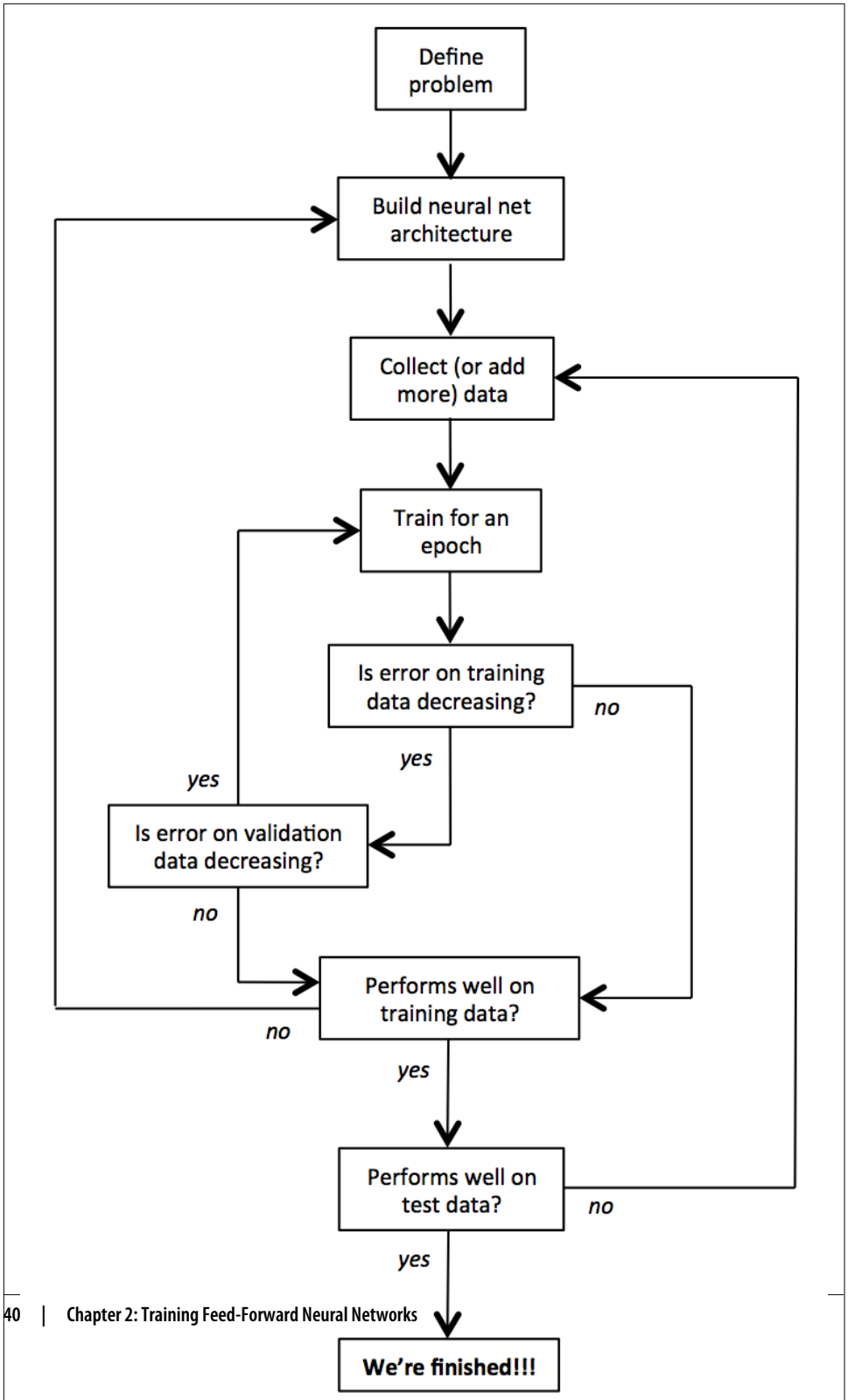
Figure 2-13. In deep learning we often include a validation set to prevent overfitting during the training process.

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

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

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.


Define problem
Build neural net architecture
Collect (or add more) data
Train for an epoch
Is error on training data decreasing?
no
yes
yes
Is error on validation data decreasing?
no
Performs well on training data?
no
yes
Performs well on test data?
no
yes
We're finished!!!

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 $Error + \lambda f(\theta)$, 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**.

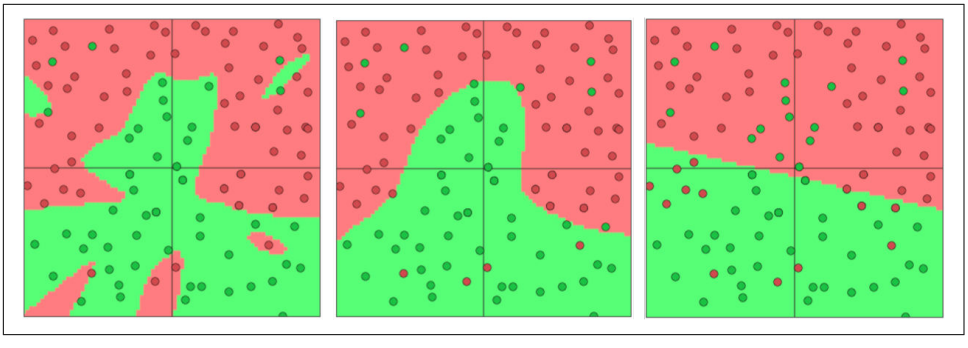


Figure 2-15. A visualization of neural networks trained with regularization strengths of 0.01, 0.1, and 1 (in that order).

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**.

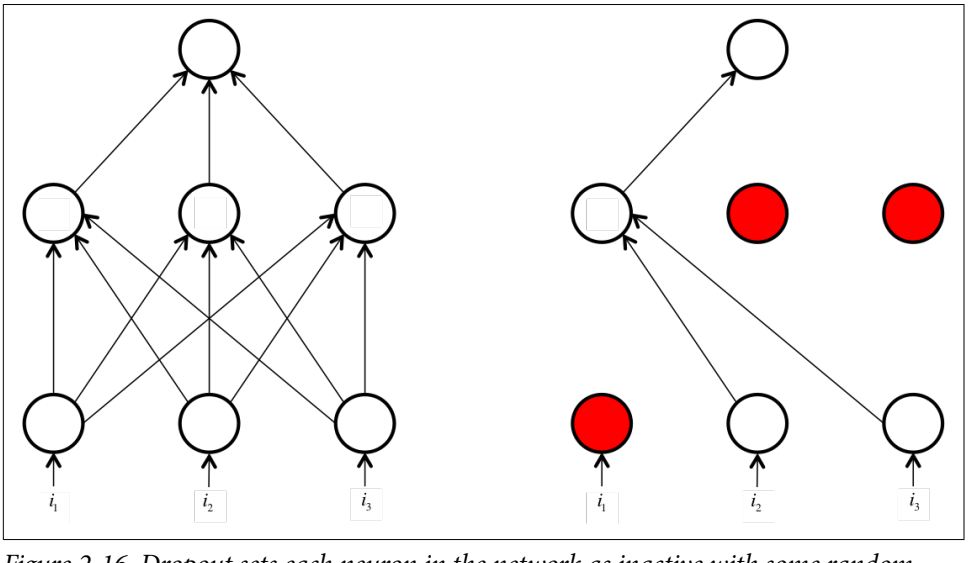


Figure 2-16. Dropout sets each neuron in the network as inactive with some random probability during each mini-batch of training.

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, Layer) + 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 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, Layer) + 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.

CHAPTER 3

Implementing Neural Networks in TensorFlow

What is TensorFlow?

Although we could spend this entire book describing deep learning models in the abstract, we hope that by the end of this text, you not only have an understanding of how deep models work, but 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**.

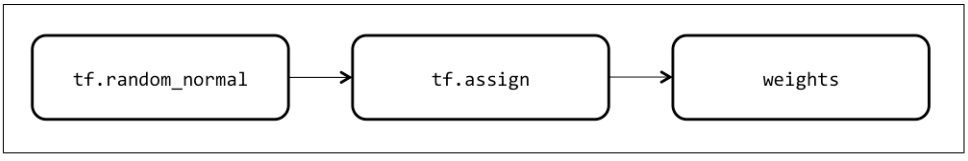


Figure 3-1. Three operations are added to the graph when instantiating a TensorFlow variable. In this example, we instantiate the variable weights using a random normal initializer.

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.

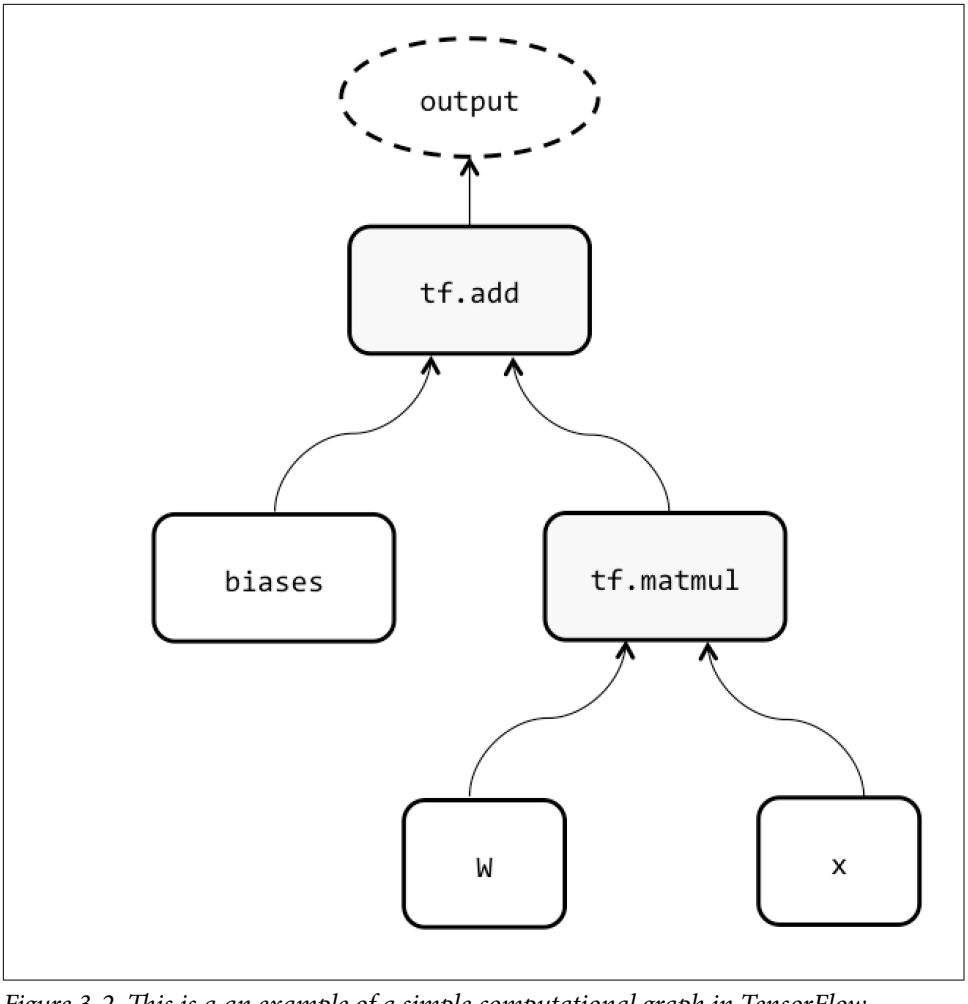


Figure 3-2. This is a an example of a simple computational graph in TensorFlow

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)
```

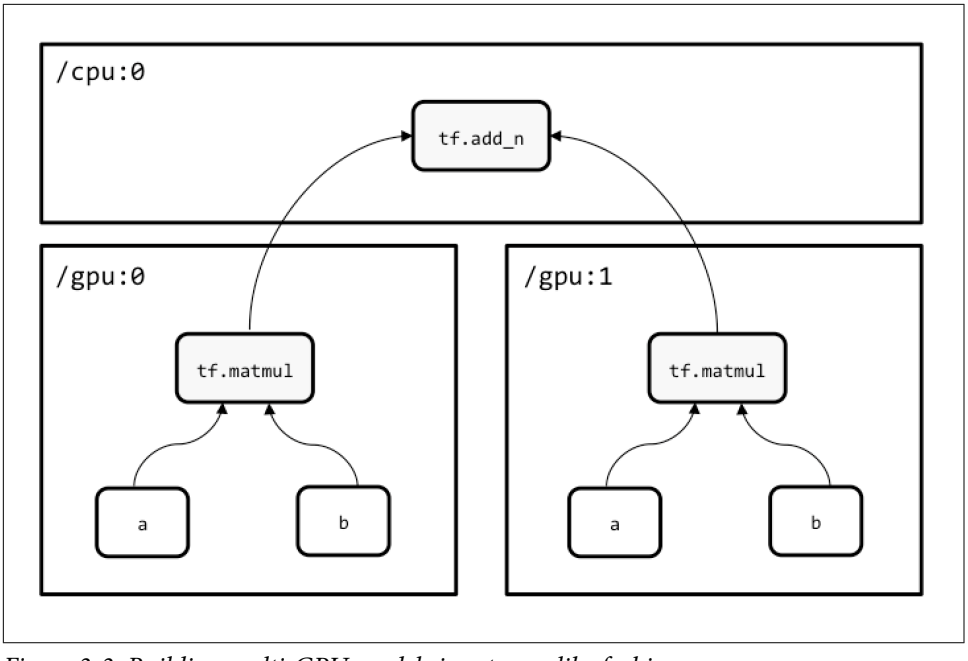


Figure 3-3. Building multi-GPU models in a tower-like fashion

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:

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

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

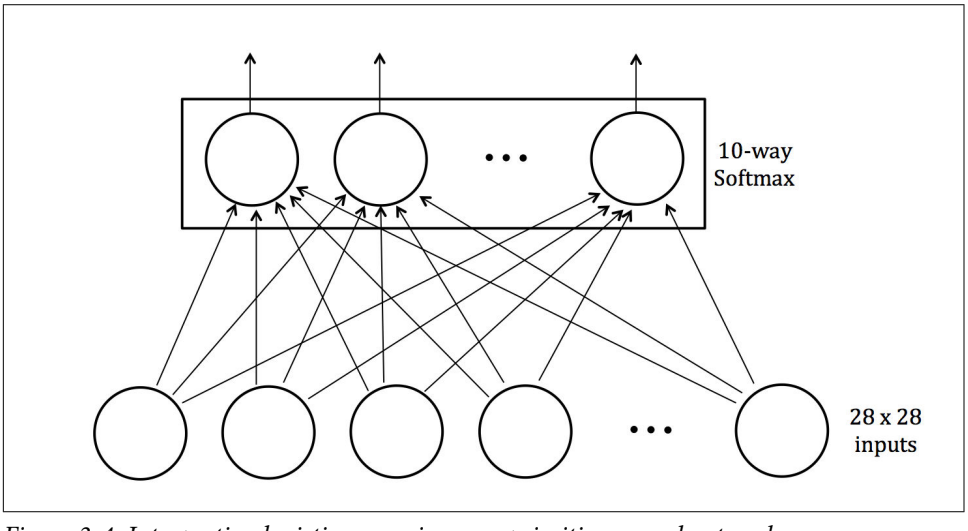


Figure 3-4. Interpreting logistic regression as a primitive neural network

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 minibatches 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.

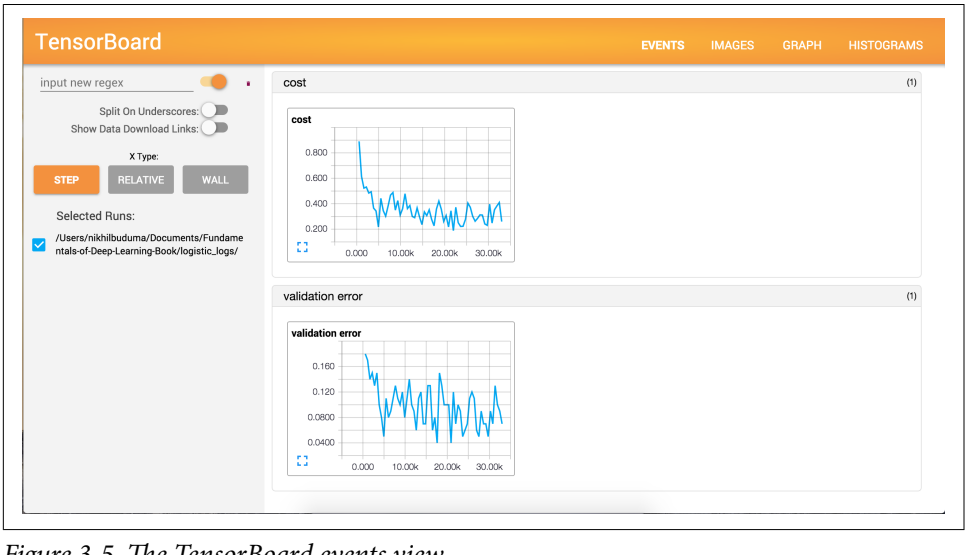


Figure 3-5. The TensorBoard events view

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

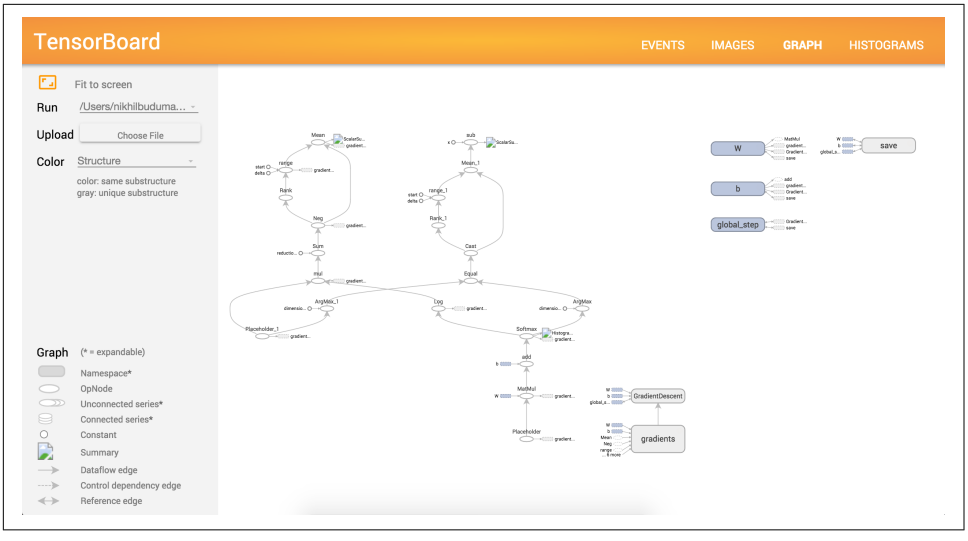


Figure 3-6. The TensorBoard graph view

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.

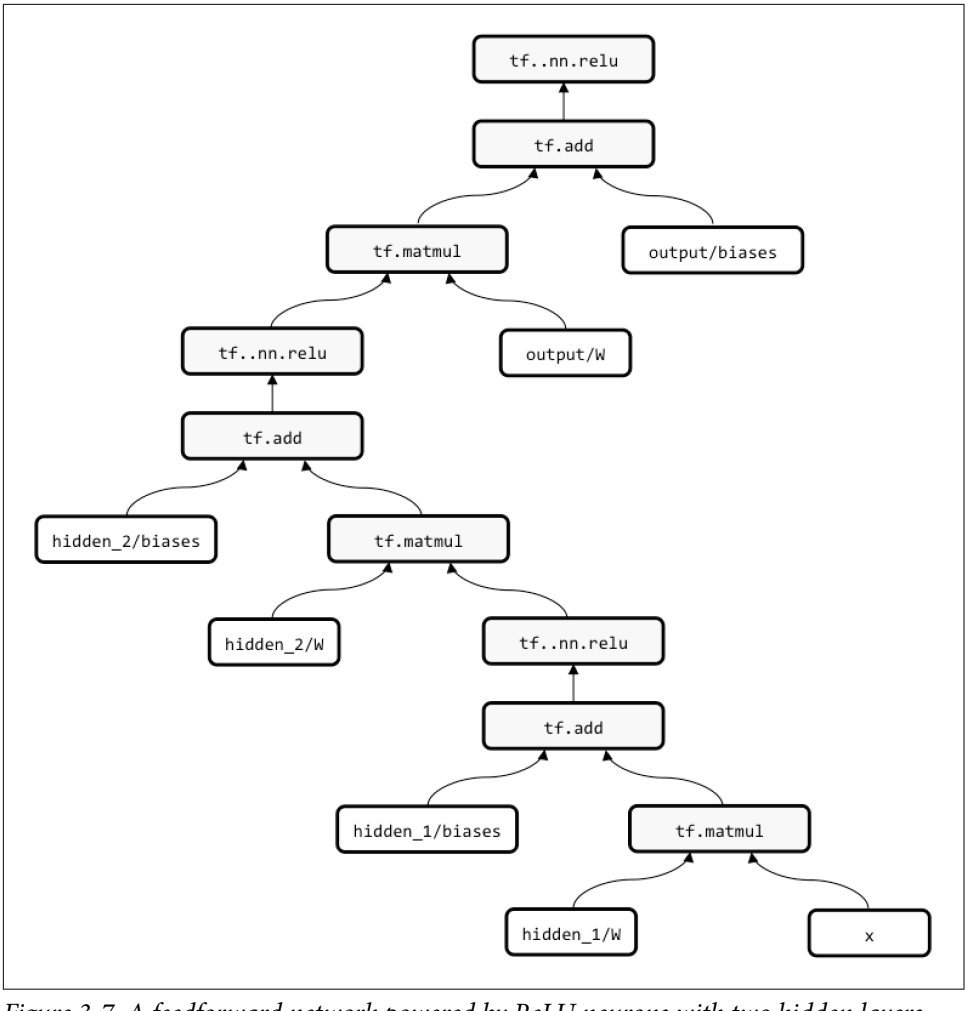


Figure 3-7. A feedforward network powered by ReLU neurons with two hidden layers

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 $\frac{2}{n_{\text{in}}}$, 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.

About the Author

Nikhil Buduma is a computer science student at MIT with deep interests in machine learning and the biomedical sciences. He is a two time gold medalist at the International Biology Olympiad, a student researcher, and a "hacker." He was selected as a finalist in the 2012 International BioGENEius Challenge for his research on the pertussis vaccine, and served as the lab manager of the Veregge Lab at San Jose State University at the age of 16. At age 19, he had a first author publication on using protist models for high throughput drug screening using flow cytometry. Nikhil also has a passion for education, regularly writing technical posts on his blog, teaching machine learning tutorials at hackathons, and recently, received the Young Innovator Award from the Gordon and Betty Moore Foundation for re-invisioning the traditional chemistry set using augmented reality.