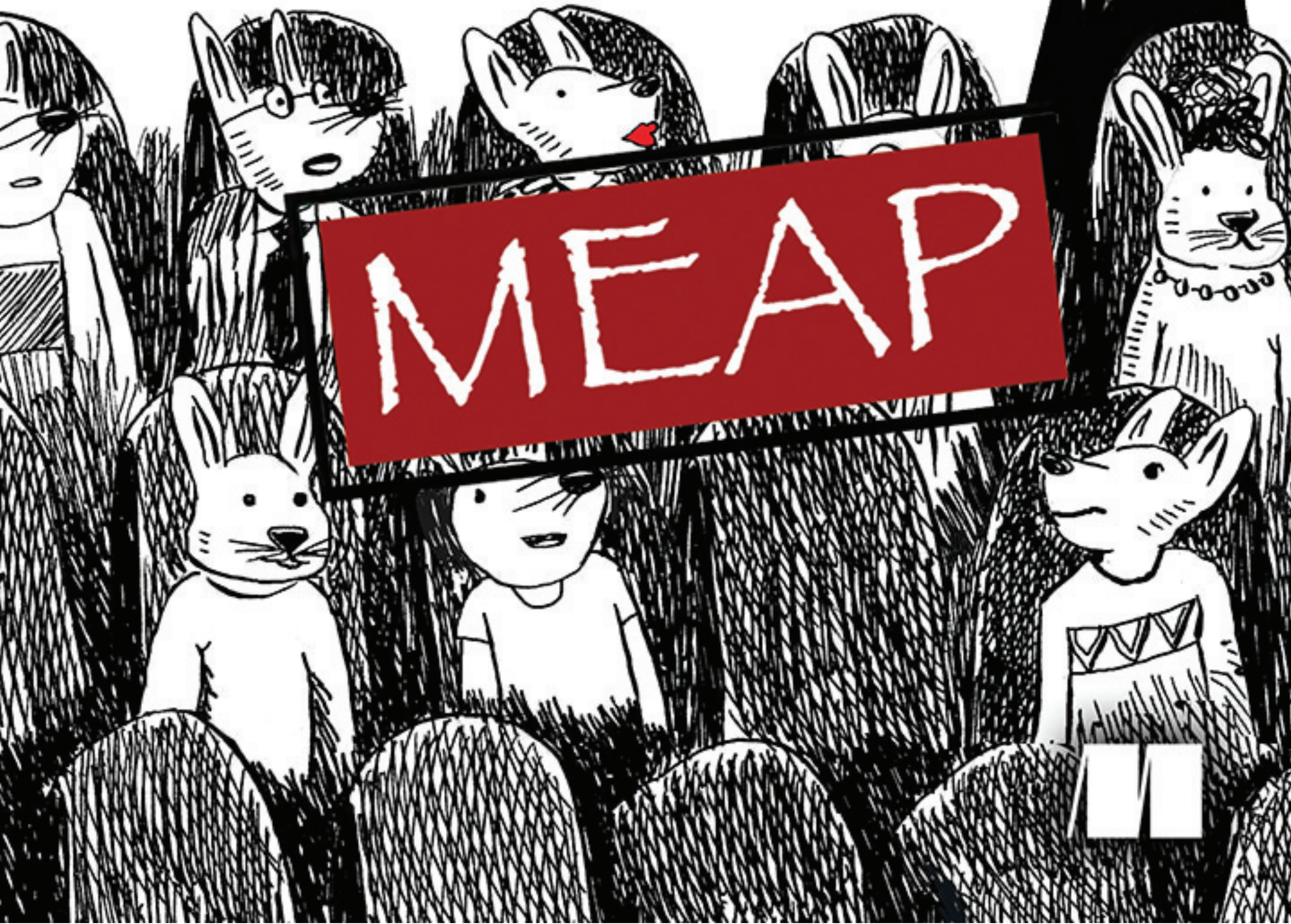
grokking

Deep Learning

Andrew W. Trask



MEAP Edition
Manning Early Access Program

Grokking Deep Learning

Version 3

welcome

Thank you so much for purchasing *Grokking Deep Learning*. This book will teach you the fundamentals of Deep Learning from an intuitive perspective, so that you can understand how machines learn using Deep Learning. This book is not focused on learning a framework such as Torch, TensorFlow, or Keras. Instead, it is focused on teaching you the Deep Learning methods behind well known frameworks. Everything will be built from scratch using only Python and numpy (a matrix library). In this way, you will understand every detail that goes into training a neural network, not just how to use a code library. You should consider this book a pre-requisite to mastering one of the major frameworks.

There are many other resources for learning Deep Learning. I'm glad that you came to this one, as I have intentionally written it with what I believe is the lowest barrier to entry possible. No knowledge of Linear Algebra, Calculus, Convex Optimization, or even Machine Learning is assumed. Everything from these subjects that is necessary to understand Deep Learning will be explained as we go. If you have passed high school mathematics and hacked around in python, you're ready for this book, and when you complete this book, you will be ready to master a major deep learning framework.

Finally, as this is the MEAP, if there is any point in these first few chapters that something does not make sense, it is my hope that you would tweet your questions to me @iamtrask. I would be happy to help, and more importantly, I want to know if any section of the book is not fulfilling my personal commitment to the lowest barrier to entry possible so that I can adjust it for the final published work. Please, don't hesitate to reach out if you have questions.

These first three chapters will be walking you from a general introduction to Deep Learning all the way through to building your first working neural network. In these chapters, you will get a firm grasp on the philosophy behind how machines can learn the world you present to them. It's an exciting thing to see happen, and perhaps even more exciting, you will understand every nook and cranny of what makes this learning possible.

It is an honor to have your time and attention.
—Andrew Trask

brief contents

Introducing Deep Learning
Why you should learn it

1

IN THIS CHAPTER

Why you should learn deep learning

Why you should read this book

What you need to get started

Welcome to Grokking Deep Learning

You're about to learn some of the most valuable skills of the century!

I'm very excited that you're here! You should be too! Deep Learning represents an exciting intersection of Machine Learning and Artificial Intelligence and a very significant impact on society and industry. The methods discussed in this book are changing the world all around you. From optimizing the engine of your car to deciding which content you view on social media, it's everywhere. It's powerful. And quite frankly, it's fun!

Why should you learn Deep Learning?

It is a powerful tool for the *incremental automation of intelligence.*

From the beginning of time, humans have been building better and better tools to understand and control the environment around us. Deep Learning is today's chapter in this story of innovation. Perhaps what makes this chapter so compelling is that this field is more of a *mental* innovation than a *mechanical one*. Much like its sister fields in Machine Learning, Deep Learning seeks to *automate intelligence* bit by bit, and in the past few years it has achieved enourmous success and progress in this endeavor, exceeding previous records in Computer Vision, Speech Recognition, Machine Translation, and many other tasks. This is particularly extraordinary given that Deep Learning seems to use *largely the same brain-inspired algorithm* (Neural Networks) for achieving these accomplishments across a vast number of fields. This has lead to tremendous excitement that perhaps we have in fact discovered more than just a great tool, but a window into our own minds as well.

Deep Learning has the potential for significant automation of *skilled labor.*

There is a substantial amount of hype around the potential impacts of Deep Learning if the current trend of progress is extrapolated at varying speeds. While many of these predictions are over-zealous, there is one that I think merits your consideration, job displacement. I think that this claim stands out from the rest for no other reason than if Deep Learning's innovations stopped *today*, there would already be an incredible impact on skilled labor around the globe. Call center operators, taxi drivers, and low-level business analysts are compelling examples where Deep Learning can provide a low-cost alternative. Fortunately, the economy doesn't turn on a dime, but in many ways we are already past the point of concern with the current power of the technology. It is my hope that you (and people you know) will be enabled by this book to transition from perhaps one of the industries facing disruption into an industry ripe with growth and prosperity, Deep Learning.

It's fun and incredibly creative. You will discover much about what it is to be human by trying to simulate intelligence and creativity.

Personally, I got into Deep Learning because it's fascinating. It's an amazing intersection between man and machine. Unpacking exactly what it means to think, to reason, and to create is enlightening, engaging, and for me it's quite inspiring. Consider having a dataset filled with every painting ever painted, and then using that to teach a machine how to paint like Monet. Insanely, it's possible, and it's mind-bogglingly cool to see how it works.

Why you should read this book!

Uniquely low barrier to entry

The reason you should read this book is the same reason I'm writing it. I don't know of another resource (book, course, large blog series) that teaches Deep Learning **without assuming advanced knowledge of mathematics** (i.e. college degree in a mathy field). Don't get me wrong, there are really good reasons for teaching it using math. Math is, after all, a language. It is certainly more **efficient** to teach Deep Learning using this language, but I don't think it's absolutely necessary to assume advanced knowledge of math in order to become a skilled, knowledgable practitioner who has a firm understanding of the "how" behind Deep Learning. So, why should you learn Deep Learning using this book? I'm going to assume you have a High School level background in math (and that it's rusty), and *explain everything else you need to know as we go along.* Remember multiplication? Remember x-y graphs (the square with lines on it)? Awesome! You'll be fine.

To help you understand what's *inside* a framework (Torch, TensorFlow, etc.)

There are two major groups of Deep Learning educational material (books, courses, etc.). One group is focused around how to use popular frameworks and code libraries such as Torch, Tensorflow, Keras, and others. The other group is focused around teaching Deep Learning itself, otherwise known as the *science under the hood* of these major frameworks. Ultimately, learning about *both* is important. It's like if you want to be a NASCAR driver, you need to learn BOTH about the particular model of car you're driving (the framework), AND about driving itself (the science/skill). However, just learning about a framework is like learning about the pros and cons of a Generation-6 Chevrolet SS before you know what a stick shift is. This book is about teaching you what *Deep Learning* is so that you can then learn a framework.

All math related material will be backed by intuitive *analogies*.

Whenever I encounter a math formula in the wild, I take a two-step approach. The first is to translate its methods into an intuitive *analogy* to the real world. I almost never just take a formula at face value. I break it into *parts*, each with a story of its own. That will be the approach of this book as well. Anytime we encounter a math concept, I'll offer an alternative *analogy* for what the formula is actually doing.

"Everything should be made as simple as possible, but no simpler"
- Albert Einstein

Everything after the introduction chapters is "project" based.

If there is one thing I hate when learning something new, it is when I have to question whether or not what I'm learning is useful/relevant. If someone is teaching me everything there is to know about a hammer without actually taking my hand and helping me drive in a nail, then they're not really teaching me how to use a hammer. I know that there are going to be dots that weren't connected, and if I was thrown out into the real world with a hammer, a box of nails, and a bunch of 2x4s, I'm going to have to do some guesswork.

This book is about giving you the wood, nails, and a hammer *before* telling you about what they do. Each lesson is about picking up the tools and building stuff with them, explaining how stuff works along the way. In this way, you don't leave with a list of facts about the various deep learning tools we'll work with, you leave with the ability to use them to solve problems. Furthermore, you will understand the most important part, when and why each tool is appropriate for each problem you want to solve. It is with this knowledge that you will be empowered to pursue a career in research and/or industry.

What you need to get started

Install Jupyter Notebook and the Numpy python library

My absolute favorite place to work is a Jupyter Notebook. One of the most important parts of learning deep learning (for me), is the abiliy to stop a network while it's training and tear apart absolutely every piece and see what it looks like. This is something that jupyter notebook is incredibly useful for. As for numpy, perhaps the most compelling case for why this book leaves nothing out is that we'll only be using a single matrix library. In this way, you will understand **how** everything works, not just how to call a framework. This book teaches Deep Learning from absolute scratch.... soup to nuts. Installation instructions for these two tools can be found at (http://jupyter.org/) for Jupyter and (http://numpy.org) for numpy.

Pass High School Mathematics

There are some mathematical assumptions that are simply out of depth for this book, but the goal of this book is to teach Deep Learning assuming you understand basic algebra.

Find a personal problem you are interested in

This might seem like an optional "need" to get started. I guess it could be, but seriously, I highly, highly recommend finding one. Everyone I know who has become successful at this stuff had some sort of problem they were trying to solve. Learning this stuff was just a "dependency" to solving some other interesting task. For me, it was using Twitter to predict the stock market. It's just something that I thought was really fascinating. It's what drove me to sit down and read the next chapter and build the next prototype. And as it turns out, this field is **so new**... and is changing **so fast**... that if you spend the next couple years chasing one project with these tools, you'll find yourself being one of the leading experts in that *particular problem* faster than you might think. For me, chasing this idea took me from barely knowing anything about programming to a research grant at a hedge fund applying what I learned in around 18 months! Having a problem you're fascinated with that involves using one dataset to predict another is the key catalyst! Go find one!

You'll probably need some Python knowledge

Python is my teaching library of choice, but I'll provide a few others online.

Python is an amazingly intuitive language. I think it just might be the most widely adopted and intuitively readable language yet constructed. Furthermore, the Python community has a passion for simplicity that can't be beat. For these reasons, I want to stick with python for all of the examples. On this book's Github, I'll provide all of the examples in a variety of other languages as well, but for the in-page explanations, we're going to use python.

How much coding experience should you have? At least the basics...
Scan through the Python Codecademy course (https://www.codecademy.com/learn/python). If you can read through the table of contents and feel comfortable with the terms mentioned, you're all set! If not, then just take the course and come back when you're done! It's designed to be a beginner course and it's very well crafted.

Conclusion and Primer for Chapter 2

So, if you've got your Jupyter Notebook in-hand and feel comfortable with the basics of Python, you're ready for the next chapter! As a heads up, Chapter 2 is the last chapter that will be mostly dialogue based (without building something). It's just designed to give you an awareness of the high level vocabulary, concepts, and fields in Artificial Intelligence, Machine Learning, and most importantly... Deep Learning.

Fundamental Concepts
How do machines learn?

2

IN THIS CHAPTER

What are Deep Learning, Machine Learning, and Artificial Intelligence?

What is a Parametric Model?

What is a Non-Parametric Model?

What is Supervised Learning?

What is Unsupervised Learning?

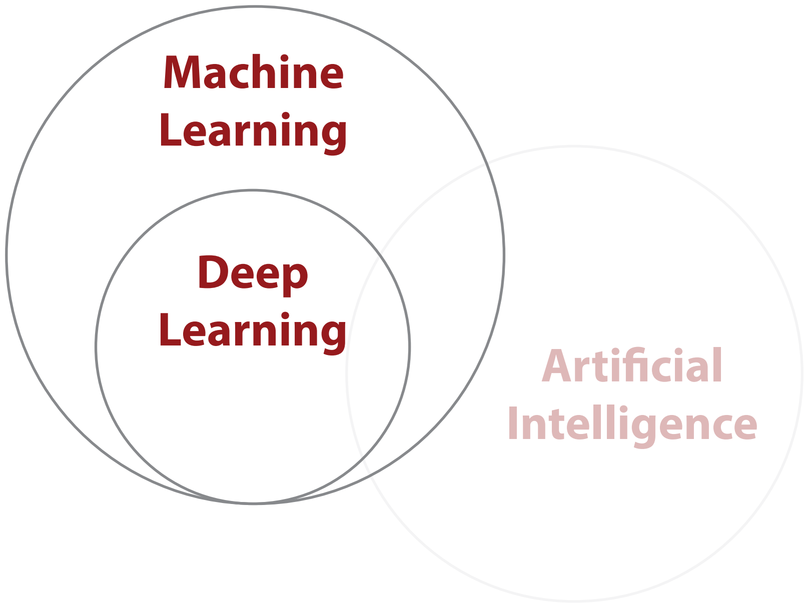
How can machines learn?

“Machine Learning will cause every successful IPO win in 5 years”
- Eric Schmidt (Google CEO)

What is Deep Learning?

Deep Learning is a subfield of methods for Machine Learning

Deep Learning is a subset of Machine Learning, which is a field dedicated to the study and development of machines that can learn (sometimes with the goal of eventually attaining General Artificial Intelligence). In industry, Deep Learning is used to solve practical tasks in a variety of fields such as Computer Vision (Image), Natural Language Processing (Text), and Automatic Speech Recognition (Audio). In short, Deep Learning is a subset of *methods* in the Machine Learning toolbox, primarily leveraging **Artificial Neural Networks**, which are a class of algorithm loosely inspired by the human brain.



Notice in the figure above that not all of Deep Learning is focused around pursuing Generalized Artificial Intelligence. In fact, many applications of this technology are appled to solve a wide variety of problems in industry. As an aside, this book seeks to focus on learning the fundamentals of Deep Learning behind both cutting edge research and industry. I will help prepare you for both.

What is Machine Learning?

"A field of study that gives computers the ability to learn without being explicitly programmed'"

- Arthur Samuel

Given that Deep Learning is a subset of Machine Learning, what is Machine Learning? Most generally, it is what its name implies. Machine Learning is a subfield of Computer Science wherein *machines learn* to perform tasks for which they were *not explicitly programmed*. In short, machines observe a pattern and attempt to immitate it in some way which can be either direct or indirect **imitation**.

machine learning ~= monkey see monkey do

I mention direct and indirect imitation as a parallel to the two main types of machine learning, **supervised** machine learning and **unsupervised** machine learning. Supervised machine learning is the direct imitation of a pattern between two datasets. It is always attempting to take an input dataset and transform it into an output dataset. This can be an incredibly powerful and useful capability. Consider the following examples: (**input** datasets in bold and *output* datasets in italic)

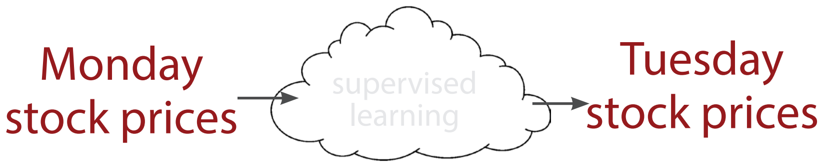
- Using the **pixels** of an image to detect the *presence or absence of a cat*
- Using the **movies you've liked** to predict *movies you may like*
- Using someone's **words** to predict whether they are *happy* or *sad.*
- Using weather sensor **data** to predict the *probability of rain.*
- Using car engine **sensors** to predict the optimal tuning *settings.*
- Using news **data** to predict tomorrow's stock **price**.
- Using an input **number** to predict a *number* double its size.
- Using a raw **audio file** to predict a *transcript* of the audio.

These are all supervised machine learning tasks. In all cases the machine learning algorithm is attempting to imitate the pattern between the two datasets in such a way that it can **use one dataset to predict the other**. For any example above, imagine if you had the power to predict the *output* dataset given only the **input** dataset. Pretty incredible!

Supervised Machine Learning

Supervised Learning transforms one dataset into another.

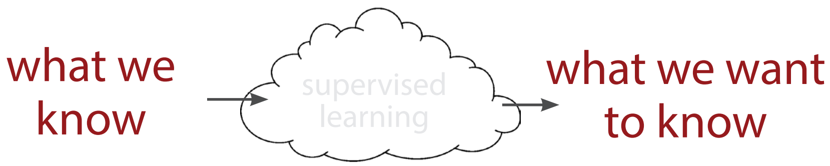
Supervised Learning is a method for transforming one dataset into another. For example, if we had a dataset of "Monday Stock Prices" which recorded the price of every stock on every Monday for the past 10 years, and a second dataset of "Tuesday Stock Prices" recorded over the same time period, a supervised learning algorithm might try to use one to predict the other.



If we successfully trained our supervised machine learning algorithm on 10 years of Mondays and Tuesdays, then we could predict the stock price of any Tuesday in the future given the stock price on the immediately preceeding Monday. I encourage you to stop and consider this for a moment.

Supervised machine learning is the bread and butter of applied Artificial Intelligence. It is useful for taking *what we do know* as input and quickly transforming it into **what we want to know**. This allows supervised machine learning algorithms to extend human intelligence and capabilities in a seemingly endless number of ways.

The majority of work leveraging machine learning results in the training of a supervised classifier of some kind. Even unsupervised machine learning (which we will learn more about in a second) is typically done to aid in the development of an accurate supervised machine learning algorithm.



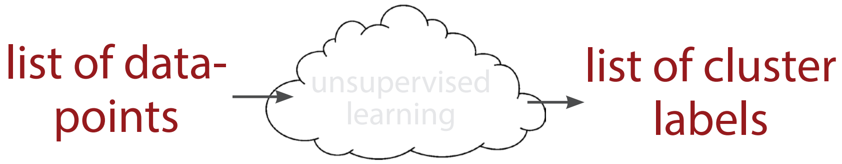
For the rest of this book, we will be creating algorithms that can take input data that is observable, recordable, and by extension **knowable** and transform it into valuable output data that requires logical analysis. This is the power of supervised machine learning.

Unsupervised Machine Learning

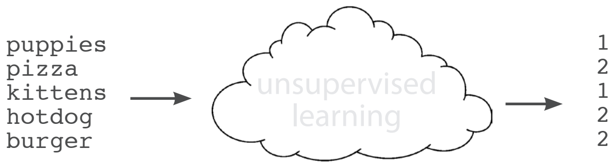
Unsupervised Learning groups your data.

Unsupervised learning shares a property in common with supervised learning. It transforms one dataset into another. However, the dataset that it transforms into is **not previously known or understood**. Unlike supervised learning, there is no "right answer" that we're trying to get the model to duplicate. We just tell an unsupervised algorithm to "find patterns in this data and tell me about them".

For example, *clustering a dataset into groups* is a type of unsupervised learning. "Clustering" transforms your sequence of *datapoints* into a sequence of *cluster labels*. If it learns 10 clusters, it's common for these labels to be the numbers 1-10. Each datapoint will get assigned to a number based on which cluster its in. Thus, your dataset turns from a bunch of datapoints into a bunch of labels. Why are the lables numbers? The algorithm doesn't tell us what the clusters are. How could it know? It just says "Hey scientist!... I found some structure. It looks like there are some groups in your data. Here they are!".



I have good news! This idea of clustering is something you can reliably hold onto in your mind as the definition of unsupervised learning. Even though there are many forms of unsupervised learning, *all forms of unsupervised learning can be viewed as a form of clustering.*

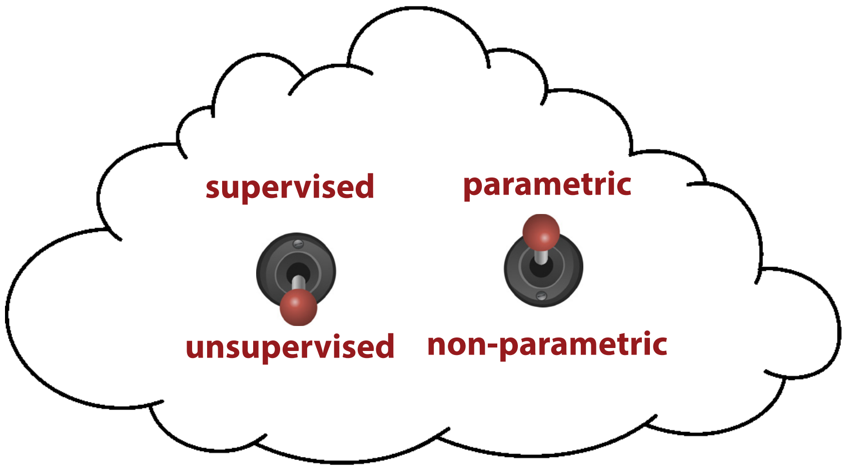


Check out the example above. Even though the algorithm didn't tell us what the clusters are named. Can you figure out how it clustered the words? (1 == cute & 2 == delicious) Later, we will unpack how other forms of unsupervsied learning are also just a forms of clustering and why these clusters are useful for supervised learning.

Parametric vs Non-Parametric Learning

Oversimplified: Trial and error learning versus counting and probability

The last two pages divided all of our machine learning algorithms into two groups, supervised and unsupervised. Now, we're going to discuss another way to divide the same machine learning algorithms into two groups, parametric and non-parametric. So, if we think about our little machine learning cloud, it has two settings:



As you can see, we really have four different types of algorithm to choose from. An algorithm is either unsupervised or supervised and it is either parametric or non-parametric. Whereas the previous section on supervision is really about the **type of pattern** being learned, parametricism is about the way the learning is **stored** and often by extension, the **method for learning**. First, let's look at the formal definition for parametricism vs non-parametricism. For the record, there is still some debate around the exact difference.

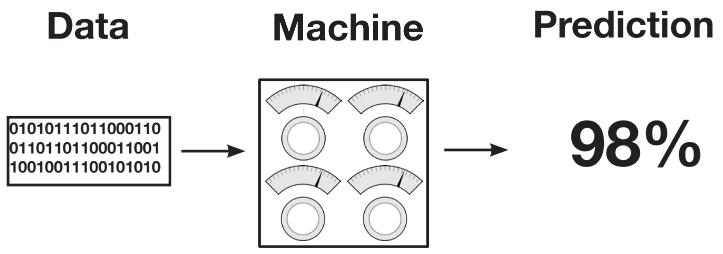
A parametric model is characterized by having a fixed number of parameters whereas a non-parametric model's number of parameters is infinite (determined by data).

Let's consider two different styles in which humans learn. Let's say the problem was to fit a square peg into the correct (square) hole. Some humans (such as babies) just jam it into all the holes until it fits somewhere. A teenager, however, might just count the number of sides (4) and then search for the hole with an equal number. This is based on his previous knowledge (valid assumption) about shapes.

Supervised Parametric Learning

Oversimplified: Trial and error learning using knobs

Supervised parametric learning machines are machines with a fixed number of knobs (that's the parametric part), wherein learning occurs by turning the knobs. **Input data** comes in, is processed based on the angle of the knobs, and is transformed into a *prediction*.



Learning is accomplished by turning the knobs to different angles. If we're trying to predict the probability that the Red Socks will win the World Series, then this model would first take data (such as sports stats like win/loss record or average number of toes) and make a prediction (such as 98% chance). Next, the model would observe whether or not the Red Socks actually won. After it knew whether they won, our learning algorithm would **update the knobs** to make a more accurate prediction the next time it sees the **same/similar input data.**

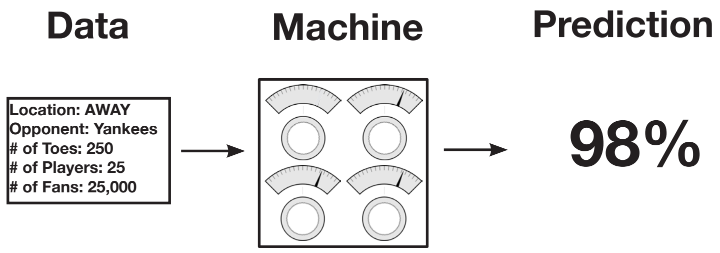
Perhaps it would "turn up" the "win/loss record" knob if the team's win/loss record was a good predictor. Inversely, it might turn down the "average number of toes" knob if that datapoint wasn't a good predictor. This is how parametric models learn!

Note that the entirety of what the model has learned can be captured in the positions of the nobs at any given time. One can also think of this type of learning model as a search algorithm. We are "searching" for the appropriate nob configuration by trying configurations, adjusting them, and retrying.

Note further that the notion of trial and error isn't the formal definition, but it is a very common (with exceptions) property to parametric models. When there is an arbitrary (but fixed) number of nobs to turn, then it requires some level of searching to find the optimal configuration. This is in contrast to non-parametric learning, which is often "count" based and (more or less) "adds new knobs" when it finds something new to count. Let's break down supervised parametric learning.

Step 1: Predict

To illustrate supervised parametric learning, let's continue with our sports analogy where we're trying to predict whether or not the Red Socks will win the World Series. The first step, as mentioned, is to gather sports statistics, send them through our machine, and make a prediction on the probability that the Red Socks will win.



Step 2: **Compare** to Truth Pattern

The second step is to compare the prediction (98%) with the pattern that we care about (whether the Red Socks won). Sadly, they lost, so our comparison is:

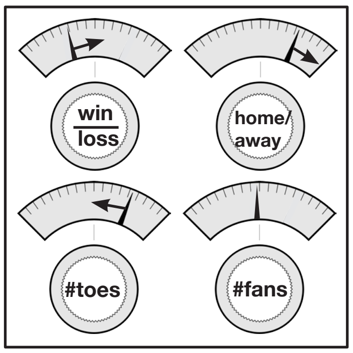
Pred: **98%** > Truth: **0%**

This step simply recognizes that if our model had predicted 0%, it would have perfectly predicted the upcoming loss of the team. We want our machine to be accurate, which takes us to Step 3.

Step 3: **Learn** the Pattern

This step adjusts the nobs by studying both how **much** the model missed (98%) and what the input data **was** (sports stats) at the time of prediction. It then turns the nobs to make a more accurate prediction given the input data. In theory, the next time it saw the same sports stats, the prediction would be lower than 98%. Note that each knob represents the *prediction's sensitivity to different types of input data*. That's what we're changing when we "learn".

Adjusting Sensitivity By Turning Knobs



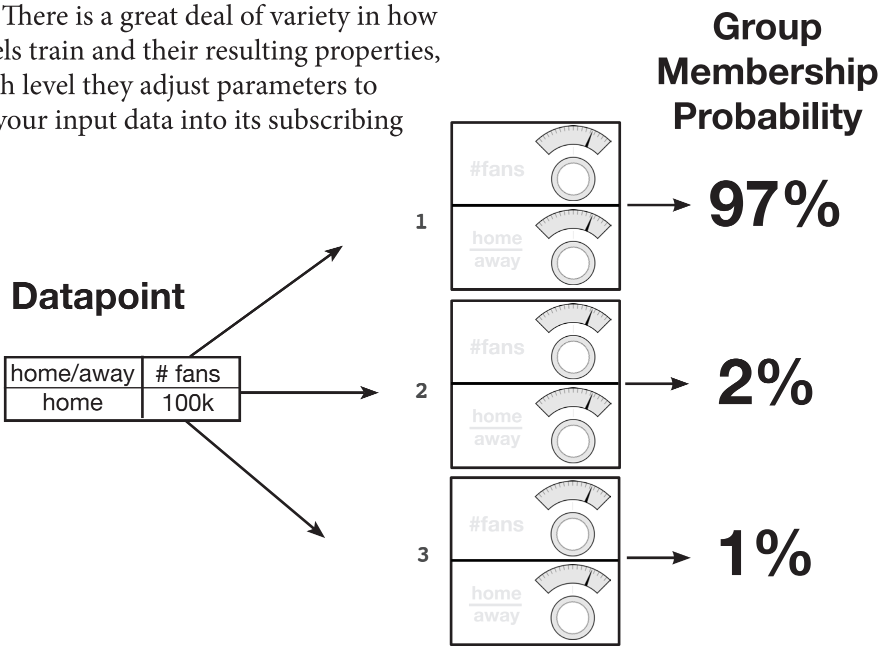
Unsupervised Parametric Learning

Unsupervised parametric learning leverages a very similar approach. Let's walk through the steps at a high level. Remember that unsupervised learning is all about grouping your data. Unsupervised *parametric* learning uses knobs to group your data. However, in this case, it usually has several knobs for each group, each that map your input data's affinity to that particular group (with exception and nuance, this is a high level description). Let's look at an example where we assume we want to divide our data into three groups.

home/away	# fans
home	**100k**
away	50k
home	**100k**
home	**99k**
away	50k
away	*10k*
away	*11k*

In the dataset on the right, I have identified three clusters in the data that we might want our parametric model to find. I identified them via formatting as **group 1**, group 2, and *group 3*. Let's propagate our first datapoint through a trained unsupervised model below. Notice that it maps most strongly to **group one**.

Each group's machine attempts to transform the input data to a number between 0 and 1, telling us the *probability that the input data is a member of that group*. There is a great deal of variety in how these models train and their resulting properties, but at a high level they adjust parameters to transform your input data into its subscribing group(s).



Conclusion

Deep Learning is Parametric
Deep Learning leverages Neural Networks to perform both supervised and unsupervised prediction. Neural Networks are incredibly powerful **parametric** models that transform your input data into output data using a combination of matrices and differentiable functions.

It should now be apparent what we're going to do. We're going to try to predict datasets given other datasets (Supervised Machine Learning). We're going to do so by creating random functions and then tweaking them to best make our predictions (Parametric Models). We're also going to learn how to induce structure in our data (Unsupervised Learning). And for all of this, we're going to be using Deep Learning (Neural Networks). In the next section, we're going to build our first neural net!

Building Your First Neural Network
Introduction to Gradient Descent 3

IN THIS CHAPTER

What is a Neural Network?

What does a Neural Network do?

Does the network make accurate predictions?

Why measure error?

What's the simplest form of neural network learning?

Calculating both *direction* and *amount* from *error*

Learning is Just Reducing Error

"This is a sample quote that I'm trying to make, but Andrew... would you like to perhaps put something else here?"

— *SOME AUTHOR*

What am I going to learn in this chapter?

Here are the terms and lessons to expect.
So, you're about to build your first neural network. Before we jump into it, I want to just tell you the lessons we're going to flesh out in this chapter. Much of this will not make *complete* sense to you just yet, and that's ok. I just want you to read these lessons at a high level so that you know the right *vocabulary*, and general *ideas* to expect when you're reading. I hope you come away with *questions* in your mind, which will help you recognize *answers* while you're reading.

What is a function?
Think about when you create a new function or method in Python. It takes some parameters as input and it returns some sort of value. Math functions are no different.

What is a neural network?
It's just a function. In python-speak, it looks like this.

```
def neural_network(input_data, weight):

    prediction = input * knob_weight

    return prediction
```

Neural networks can have more weights, more inputs, and more outputs than this network above, but there are *always* two parameters to the function, (input_data, weights), and one output (prediction). These variables just become lists (or lists of lists) when we have more of them. (More on that later) The important takeaway at the moment is the API of a neural network. Data and weights go in... predictions come out.

What does a neural network do?
A neural network *learns a function*. This might seem confusing since I just told you that it *is a funtion*. However, every neural network starts out predicting *randomly*. In other words, our starting *weight* values are random... thus our function predicts *randomly*. It's a random function.

As you may remember from the previous chapter, a neural network learns how to take an *input dataset* and convert it into an *output dataset*. For example, it might take an *input dataset* of Farenheit temperatures and learn to convert it into an *output dataset* of Celsius temperatures. It might covert a *pixel values dataset (still just numbers... pixels)* into a dataset of "probability there's a cat in the photo". Both are just taking one set of numbers and converting them into another set of numbers. To a neural network... it's all the same. It's just looking for a pattern between two sets of numbers so that you can take one and use it to create the other.

How does a neural network learn?

Well, we have a second function (the more important one) called the error function (sometimes called the loss function). It looks like this:

```
def compute_error(input_data, weights, goal_prediction):

    prediction = neural_network(input_data,weights)

    raw_error = prediction - goal_prediction
    normalized_error = raw_error ** 2

    return normalized_error
```

This is the only function that really matters. "Learning" is just about modifying *weights* so that the output to this function becomes zero. Why? An error of 0 means we predicted perfectly! So, we try lots of different *weights* to figure out how to predict with 0 error.

How does one find the right weights?

```
error = compute_error(input_data, weights, goal_prediction)
```

You might notice that the error function is a pretty clean API... a well defined problem. We're really just trying to mess with `weights` so that the `error` goes to 0. For now, we might NOT even care what happens inside this `compute_error` function... we only really care about the *weights* variable, and the `error` variable.

So, how do we find the correct *weights* variable? One could just try random weights over and over again... but this turns out to just take too darn long. Instead, we can do things like "wiggle" each weight to see if it moves the error up or down... and then go whichever direction moves the error downward. Repeating this over and over ends up working pretty well.

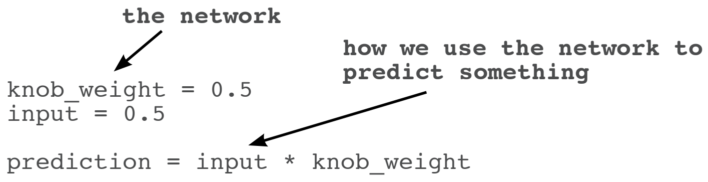
We learn by just wiggling?... are you kidding me?

That's certainly not the *fastest* or most *efficient* form of learning, but it does work. There is, however, a better method. It turns out... "wiggling" just tells us a property of our error function... the **amount** and **direction** that the `error` changes when we move any particular weight.

The compute_error function is actually an exact *mathematical relationship* between our `weights` and our `error`. It turns out that for any function, we can actually compute how much one variable (say... one particular weight) causes another variable (say...the `error`) to change. If we know how one variable causes another to change... then we have everything we need to move our `error` down to 0. We just take each weight... compute its affect on the `error`... and move it in the right direction so that the `error` goes down (to 0). That process is called **Gradient Descent**. Ok... now that we've seen the high-level *super fast* version of this chapter... let's dive in.

What is a Neural Network?

This is a neural network.
Open up your iPython Notebook. Take this code. Type it into a cell. Execute it.



```
knob_weight = 0.5
input = 0.5

prediction = input * knob_weight
```

You just made your first neural network and used it to predict! Congratulations! Print out `prediction` and see the output. It should be 0.25. So what is a neural network? It's one or more *weights* which we can multiply by our `input` data to make a `prediction`.

What is `input` data?
It's a number that we recorded in the real world somewhere. It's usually something that is easily knowable, like today's temperature, a baseball player's batting average, or yesterday's stock price.

What is a `prediction`?
A `prediction` is what the neural network tells us *given our input data* such as "given the temperature, it is **0%** likely that people will wear sweatsuits today" or "given a baseball player's batting average, he is **30%** likely to hit a home run" or "given yesterday's stock price, today's stock price will be **101.52**".

Is this `prediction` always right?
No. Sometimes our neural network will make mistakes, but it can learn from them. For example, if it predicts too high, it will adjust it's `knob_weight` to predict lower next time and vise versa.

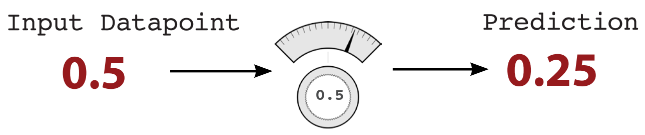
How does the network learn?
Trial and error! First, it tries to make a `prediction`. Then, it sees whether it was too high or too low. Finally, it changes the `knob_weight` (up or down) to predict more accurately next time it sees the same `input`.

What does a Neural Network do?

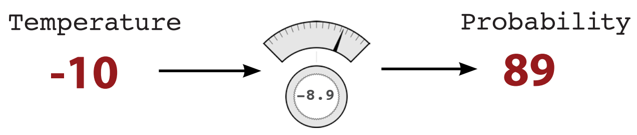
It multiplies the `input` by a `knob_weight`. It "scales" the input by a certain amount.

On the previous page, we made our first prediction with a neural network. A neural network, in it's simplest form, uses the power of *multiplication*. It takes our input datapoint (in this case, 0.5) and *multiplies* it by our `knob_weight`. If the `knob_weight` is 2, then it would *double our input*. If the `knob_weight` is 0.01, then it would *divide* the input by 100. As you can see, some `knob_weight` values make the input *bigger* and other values make it *smaller*.

Another way to think about a neural network's `knob_weight` is as a measure of *sensitivity* between the input of the network and its prediction. If the knob_weight is very high, then even the tiniest input can create a really large prediction! If the knob_weight is very small, then even large inputs will make small predictions. This *sensitivity* is very akin to ***volume***. "Turning up the weight" amplifies our prediction relative to our input. `knob_weight` is a volume knob!



Neural networks don't just predict positive numbers either, they can also *predict negative numbers*, and even take *negative numbers as input*. Perhaps you want to predict the "probably that people will wear coats today", if the temperature was -10 degrees celius, then a negative weight would predict a high probability that people would wear coats today.



Notice a few things. Our logic for prediction is the same regardless of what the network is trying to predict. The only thing that makes a neural network unique is the *value of the knob_weight*. If two neural networks have identical *weight values*, then they are identical networks! This brings us back to our definition.

> A neural network is one or more `weights` which we can *multiply* by our `input` to make a `prediction`

Does the network make accurate predictions?

Let's measure the error and find out!
Open up your iPython Notebook. Take this code. Type it into a cell. Execute it. (All modifications from the last page are in **bold**.)

what a perfect network should predict given input=0.5

"how much we missed"

"squared" i.e. "raised to the power of 2"

```
knob_weight = 0.5
input = 0.5
goal_prediction = 0.8

prediction = input * knob_weight

error = (prediction - goal_prediction) ** 2
```

You just measured how accurate your neural network is! Congratulations! The purpose of `error` is express "how much we missed" in a certain way. In this case, it's "how much we missed" multiplied by "how much we missed". In other words, it's "how much we missed" *squared*.

What is the `goal_prediction`?
Much like *input*, it's a number we recorded in the real world somewhere, but it's usually something that's hard to observe, like "the percentage of people who DID wear sweatsuits" given the temperature or "whether the batter DID in fact hit a home run" given his batting average.

Why is the error *squared*?
Think about an archer hitting a target. When he is 2 inches high, how much did he miss by? When he is two inches low, how much did he miss by? Both times he only missed by 2 inches. The primary reason why we *square* "how much we missed" is that it forces the output to be *positive*. `prediction-goal_prediction` could be negative in some situations... *unlike actual error.*

Doesn't squaring make big errors (>1) bigger and small errors (<1) smaller?
Yeah...It is kindof a weird way of measuring error... but it turns out that **amplifying** big errors and **reducing** small errors is actually ok. Later, we'll use this error to help the network learn... and we'd rather it *pay attention* to the big errors and not worry so much about the small ones. Good parents are like this too. They practically ignore errors if they're small enough (i.e. breaking the lead on your pencil) but might go nuclear for big errors (i.e. crashing the car). See why squaring is valuable?

Why measure error?

Measuring error simplifies the problem.
The goal of training our neural network is to make correct predictions. That's what we want. And in the most pragmatic world (as mentioned in the last chapter), we want the network to take input that we can easily calculate (today's stock price), and predict things that are hard to calculate (tomorrow's stock price). That's what makes a neural network useful.

It turns out that "changing knob_weight to make the network correctly predict the goal_prediction" is *slightly* more complicated than "changing the knob_weight to make error == 0" There's something more concise about looking at the problem this way. Ultimately, both of those statements say the same thing, but trying to *get the error to 0* just seems a bit more straightforward.

Different ways of measuring error *prioritize error differently*.
If this is a bit of a stretch right now, that's ok... but think back to what I said on the last page. By *squaring* the error, numbers that are less than 1 get *smaller* whereas numbers that are greater than 1 get *bigger*. This means that we're going to change what I call **"pure error" (prediction-goal_prediction)** so that bigger errors become VERY big and smaller errors quickly become irrelevant. By measuring error this way, we can *prioritize* big errors over smaller ones. When we have somewhat large "pure errors" (say... 10)... we're going to tell ourselves we have VERY large error (10**2 == 100), and in contrast, when we have small "pure errors" (say... 0.01), we're going to tell ourselves that we have VERY small error (0.01 **2 == 0.0001). See what I mean about *prioritizing*? It's just modifying what we *consider to be error* so that we amplify big ones and largely ignore small ones. In contrast, if we took the *absolute value* instead of *squaring* the error, we wouldn't have this type of prioritization. The error would just be the positive version of the "pure error"... which would be fine... but just different. More on this later.

Why do we only want *positive* error?
Eventually, we're going to be working with *millions* of input -> goal_prediction pairs... and we're still going to want to make accurate predictions. This means that we're going to try to take the *average error* down to 0.

This presents a problem if our error can be positive and negative. Imagine if we had two datapoints... two input -> goal_prediction pairs that we were trying to get the neural network to correctly predict. If the first had an error of 1,000, and the second had an error of -1,000, then our *average error* would be ZERO! We would fool ourselves into thinking we predicted perfectly when we missed by 1000 each time!!! This would be really bad. Thus, we want the error of *each prediction* to always be *positive* so that they don't accidentally cancel each other out when we average them.

What the Simplest Form of Neural Network Learning?

Perhaps the simplest form is called "hot and cold" learning.
Open up your iPython Notebook. Take this code. Type it into a cell. Execute it. (New neural network modifications are in **bold**.) This code attempts to correctly predict 0.8.

how much to move our weights each iteration

repeat learning many times so that our error can keep getting smaller

```
knob_weight = 0.5
input = 0.5
goal_prediction = 0.8

step_amount = 0.001

for iteration in range(1101):

    prediction = input * knob_weight
    error = (prediction - goal_prediction) ** 2

    print "Error:" + str(error) + " Prediction:" + str(prediction)

    up_prediction = input * (knob_weight + step_amount)
    up_error = (goal_prediction - up_prediction) ** 2

    down_prediction = input * (knob_weight - step_amount)
    down_error = (goal_prediction - down_prediction) ** 2

    if(down_error < up_error):
        knob_weight = knob_weight - step_amount

    if(down_error > up_error):
        knob_weight = knob_weight + step_amount
```

TRY UP!

TRY DOWN!

If down is better, go down!

If up is better, go up!

At the end of the day, learning is really about one thing, adjusting our **`knob_weight`** either up or down so that our **`error`** reduces. If we keep doing this and our **`error`** goes to 0, we are done learning! So, how do we know whether to turn the knob up or down? Well, we try *both up and down* and see which one reduces the error! Whichever one reduces the error we use to actually update the **`knob_weight`**. It's simple, but effective. After we do this over and over again, eventually our error==0, which means our neural network correctly predicts 0.8, achieving our goal! When I run this code, I see the following output:

```
Error:0.3025 Prediction:0.25
Error:0.30195025 Prediction:0.2505
           ....
Error:2.50000000033e-07 Prediction:0.7995
Error:1.07995057925e-27 Prediction:0.8
```

Our last step correctly predicts 0.8!

Characteristics of Hot and Cold Learning

It's simple

Hot and Cold learning is simple. After making our prediction, we predict two more times, once with a slightly higher weight and again with a slightly lower weight. We then move the `knob_weight` depending on which **direction** gave us a smaller error. Repeating this enough times eventually reduces our error down to 0.

Why did I iterate exactly 1101 times?
The neural network reaches 0.8 after exactly that many iterations. If you go past that, it wiggles back and forth between 0.8 and just above/below 0.8... making for a less pretty error log printed at the bottom of the left page. Feel free to try it out though.

PROBLEM: It's inefficient

We have to predict *multiple times* in order to make a single *knob_weight* update. This seems very inefficient.

PROBLEM: Sometimes it's impossible to predict the *exact* goal prediction.

With a set step_amount, unless the perfect knob_weight is exactly n*step_amount away, the network will eventually overshoot by some number less than step_amount. When it does so, it will then start alternating back and forth between each side of the goal_prediction. Set the step_amount to 0.2 to see this in action. If you set step_amount to 10 you'll really break it! When I try this I see the following output. It never *remotely* comes *close* to 0.8!!!

```
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
....
.... repeating infinitely...
```

The real problem here is that even though we know the correct **direction** to move our knob_weight. We *don't know the correct **amount***. Since we don't *know* the correct amount, we just pick a fixed one at random (step_amount). Furthermore, this *amount* has NOTHING to do with our error. Whether or error is BIG or our error is TINY, our step_amount is the same. So, Hot and Cold Learning is kindof a bummer... it's inefficient because we *predict 3 times for each weight update* and our *step_amount* is completely arbitrary... which can prevent us from learning the correct prediction.

What if we had a way of computing both **direction** and **amount** for each weight without having to repeatedly make predictions?

Calculating Both *direction* and *amount* from *error*

Let's measure the error and find out!
Open up your iPython Notebook. Take this code. Type it into a cell. Execute it. (All new code from the last page is in **bold**.)

(1) "pure error"

(2) scaling, negative reversal, and stopping

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5

for iteration in range(20):
    prediction = input * knob_weight
    error = (goal_prediction - prediction) ** 2
    direction_and_amount = (goal_prediction - prediction) * input
    knob_weight = knob_weight + direction_and_amount

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

What you see above is a *superior* form of learning known as **Gradient Descent**. This method allows us to (in a single line of code... seen above in **bold**) calculate both the *direction* and the *amount* that we should change our knob_weight so that we reduce our error.

What is the `direction_and_amount`?
It represents how we want to change our `knob_weight`. The first **(1)** is what we call "pure error" which equals (**`goal_prediction`** - **`prediction`**). This number represents "the raw direction and amount that we missed". The second part **(2)** is the multiplication by the **`input`** which performs scaling, negative reversal and stopping...modifying the "pure error" so that it's ready to update our `knob_weight`.

What is the "pure error"?
It's the (**`goal_prediction`** - **`prediction`**) which indicates "the raw direction and amount that we missed". If this is a *positive* number then we need to predict *higher* next time and vise versa. If this is a *big* number then we missed by a *big* amount, etc.

What is "scaling, negative reversal, and stopping"?
These three attributes have the combined affect of translating our "pure error" into "the absolute amount that we want to change our knob_weight". They do so by addressing three *major edge cases* at which points the "pure error" is not sufficient to make a good modification to our knob_weight.

What is "stopping"?
This is the first (and simplest) affect on our "pure error" caused by multiplying it by our `input`. Imagine plugging in a CD player into your stereo. If you turned the volume all the way up but the CD player was *off*... it simply wouldn't matter. "Stopping" addresses this in our neural network... if our `input` is 0, then it will force our direction_and_amount to also be 0. We don't learn (i.e. "change the volume") when our `input` is 0 because there's nothing to learn... every knob_weight value has the same error... and moving it makes no difference because the prediction is always 0.

What is "negative reversal"?
This is probably our most difficult and important affect. Normally (when input is positive), moving our knob_weight *upward* makes our prediction move *upward*. However, if our input is *negative*, then all of a sudden our knob_weight changes directions!!! When our input is *negative*, then moving our knob_weight up makes the prediction go *down*. It's reversed!!! How do we address this? Well, multiplying our "pure error" by our *input* will *reverse the sign* of our direction_and_amount in the event that our input is negative. This is "negative reversal", making our error negative if it's positive (and vise versa) if our input is negative.

What is "scaling"?
Scaling is the second affect on our "pure error" caused by multiplying it by our **`input`**. Logically, it means that if our input was big, our weight update should also be big. This is more of a "side affect" as it can often go out of control. Later, we will use *alpha* and *Batch Normalization* to address when this scaling goes out of control.

Coming Up Next: So, I've told you all about *how* **Gradient Descent** works for *this particular network*. In the following few pages we're going to address *why* it works so that you can use it to train any kind of neural network. We're also going to talk about how to modify it to make it faster, more efficient, and address edge cases to overcome weaknesses it has from time to time. For now, just run the code in the top left and observe its output. You should see the following:

Our last steps correctly approach 0.8!

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
                    ...

Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

Learning Is Just Reducing Error

Modifying knob_weight to reduce our error.
In the previous pages, you ran this code. Let's take a closer look at two lines in particular.

these lines have a *secret*

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5

for iteration in range(20):
    prediction = input * knob_weight
    error = (goal_prediction - prediction) ** 2
    direction_and_amount = (goal_prediction - prediction) * input
    knob_weight = knob_weight + direction_and_amount

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

The Golden Method for Learning

Adjusting each `knob_weight` in the correct *direction* and by the correct *amount* so that our `error` reduces to 0.

All we're trying to do is figure out the right **direction** and **amount** to modify knob_weight so that our error goes down. The secret to this lies in our prediction and error calculations. Notice that we actually use our prediction *inside* the error calculation. Let's replace our prediction variable with the code we used to generate it.

```
error = (goal_prediction - (input * knob_weight)) ** 2
```

This doesn't change the value of error at all! It just combines our two lines of code so that we compute our error directly. Now, remember that our input and our goal_prediction are actually fixed at 0.5 and 0.8 respectively (we set them before the network even starts training). So, if we replace their variables names with the values... the *secret* becomes clear

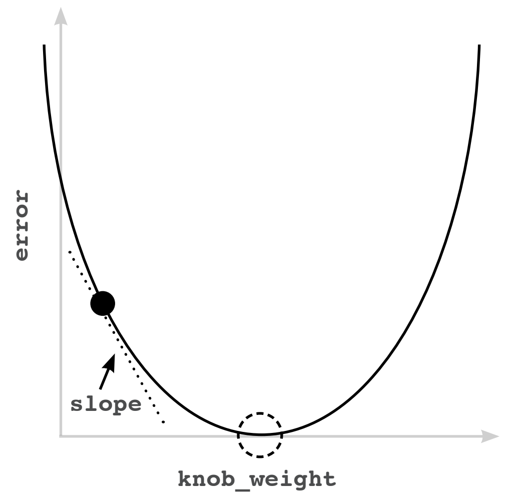
```
error = (0.8 - (0.5 * knob_weight)) ** 2
```

The Secret

For any `input` and `goal_prediction`, there is an *exact relationship* defined between our error and knob_weight, found by combining our **`prediction`** and **`error`** formulas. In this case:

```
error = (0.8 - (0.5 * knob_weight)) ** 2
```

Let's say that you moved knob_weight up by 0.5... if there is an *exact relationship* between error and knob_weight... we should be able to calculate how much this also *moves* the error! What if we wanted to *move* the error in a specific direction? Could it be done?



This graph represents *every value of error* for *every knob_weight* according to the relationship in the formula above. Notice it makes a nice *bowl shape*. The black "dot" is at the point of BOTH our current knob_weight and error. The dotted "circle" is where we want to be (error == 0).

Key Takeaway: The *slope* points to the bottom of the bowl (lowest error) *no matter where you are in the bowl*. We can use this *slope* to help our neural network *reduce the error*.

Let's Back Up And Talk about Functions

What is a function? How do we understand it?

Consider this function.

```
def my_function(x):
    return x * 2
```

A function takes some numbers as input and gives you another number as output. As you can imagine, this means that the function actually defines some sort of *relationship* between the input number(s) and the output number. Perhaps you can also see why the ability to *learn a function* is so powerful... it allows us to take some numbers (say...image pixels) and convert them into other numbers (say... the *probability* that the image contains a cat).

Now, every function has what you might call *moving parts*. It has pieces that we can tweak or change to make the ouput that the function generates *different*. Consider our "my_function" above. Ask yourself, "what is controlling the relationship between the input and the output of this function?". Well, it's the 2! Ask the same question about the function below.

```
error = (goal_prediction - (input * knob_weight)) ** 2
```

What is controlling the relationship between the **`input`** and the output (**`error`**)? Well, plenty of things are! This function is a bit more complicated! **`goal_prediction`**, **`input`**, **`**2`**, **`knob_weight`**, and all the parenthesis and algebraic operations (addition, subtraction, etc.) play a part in calculating the error... and tweaking any one of them would *change* the error. This is important to consider.

Just as a thought exercise, consider changing your **`goal_prediction`** to reduce your error. Well, this is silly... but totally doable! In life, we might call this "giving up"... setting your goals to be whatever your reality is. It's just denying that we missed! This simply wouldn't do.

What if we changed the **`input`** until our error went to zero... well... this is akin to seeing the world as you *want* to see it instead of as it actualy is. This is changing your *input data* until you're predicting what you want to predict.

Now consider changing the 2... or the additions...subtractions... or multiplications... well this is just changing how you calculate error in the first place! Our error calculation is meaningless if it doesn't actually give us a good measure of *how much we missed* (with the right properties mentioned a few pages ago). This simply won't do either.

So, what do we have left? The only variable we have left is our **`knob_weight`**. Adjusting this doesn't change our perception of the world... doesn't change our goal... and doesn't destroy our error measure. In fact... means that the function *conforms to the patterns in the data*. By forcing the rest of our function to be *unchanging*, we force our function to correctly model some pattern that exists in our data. It is only allowed to modify how the network *predicts*.

So, at the end of the day, we're modifying specific parts of an error function until the **`error`** value goes to zero. This error function is calculated using a combination of variables... some of them we can change (weights) and some of them we cannot (input data, output data, and the error logic itself).

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5

for iteration in range(20):
    prediction = input * knob_weight
    error = (goal_prediction - prediction) ** 2
    direction_and_amount = (goal_prediction - prediction) * input
    knob_weight = knob_weight + direction_and_amount

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

On the previous page, I combined the prediction and error variables into a single function. This was to show you the relationship between knob_weight and error. However, there's a very good reason that we separate them in our code.

We can modify *anything* in our **`prediction`** calculation except the **`input`**.

In fact, we're going to spend *the rest of this book* and many deep learning researchers will spend *the rest of their lives* just trying everything you can imagine to that **`prediction`** calculation so that it can make good predictions. Learning is all about automatically changing that prediction function so that it makes good predictions... aka... so that the subsequent **`error`** goes down to 0.

Ok, now that we know what we're *allowed* to change... how do we actually go about doing that changing? That's the good stuff! That's the *machine learning*, right? In the next, section, we're going to talk about exactly that.

Tunnel Vision on One Concept

Concept: "Learning is adjusting our weight to reduce the error to zero"

So far in this chapter, we've been hammering on the idea that learning is really just about adjusting our weight to reduce our error to zero. This is the secret sauce. Truth be told, knowing how to do this is *all about* understanding the **relationship** between our **`knob_weight`** and our **`error`**. If we understand this relationship, we can know how to adjust our weight to reduce our error.

What do I mean by "understand the relationship"? Well, to understand the relationship between two variables is really just to understand *how changing one variable changes the other*. In our case, what we're really after is the **sensitivity** between these two variables. Sensitivity is really just another name for *direction* and *amount*. We want to know how sensitive the error is to the knob_weight. We want to know the *direction* and the *amount* that the error changes when we change the knob_weight. This is the goal. So far, we've used two different methods to attempt to understand this relationship.

Relationship Exploration: Hot and Cold

You see, when we were "wiggling" our knob_weight and studying its affect on our error, we were really just *experimentally* studying the relationship between these two variables. It's like when you walk into a room with 15 different unlabeled light switches. You just start flipping them on and off to learn about their relationship to various lights in the room. We did the same thing to study the relationship between our **`knob_weight`** and our **`error`**. We just wiggled the **`knob_weight`** up and down and watched for how it changed the **`error`**. Once we knew the relationship, we could move the **`knob_weight`** in the right direction using a simple if statement.

```
prediction = input * knob_weight
error = (goal_prediction - prediction) ** 2
```

Check out these two lines of code from both of our learning algorithms. Notice something? They quietly define an *exact relationship* between our error and our knob_weight. Perhaps a little bit of algebra will help make it more obvious.

```
error = (goal_prediction - (input * knob_weight)) ** 2
```

This line of code, ladies and gentlemen, is the secret. This is a formula. This is the relationship between **`error`** and **`knob_weight`**.This relationship is exact. It's computable. It's universal. It is and it will always be. Now, how can we use this formula to know how to change our **`knob_weight`** so that our **`error`** moves in a *particular direction*. Now THAT is the right question! Stop. I beg you. Stop and appreciate this moment. This formula is the exact relationship between these two variables, and now we're going to figure out how to change one variable so that we move the other variable in a particular direction. As it turns out, there's a method for doing this for *any* formula. We're going to use it for reducing our error.

A Box With Rods Poking Out of It

An analogy.
Picture yourself sitting in front of a cardboard box that has two circular rods sticking through a two little holes. The blue rod is sticking out of the box by 2 inches, and the red rod is sticking out of the box by 4 inches. Imagine that I told you that these rods were connected in some way, but I wouldn't tell you in what way. You had to experiment to figure it out.

So, you take the blue rod and push it in 1 inch, and watch as... while you're pushing... the red rod also moves into the box by 2 inches!!! Then, you pull the blue rod back out an inch, and the red rod follows again!!... pulling out by 2 inches. What did you learn? Well, there seems to be a *relationship* between the red and blue rods. However much you move the blue rod, the red rod will move by twice as much. You might say the following is true.

```
red_length = blue_length * 2
```

As it turns out, there's a formal definition for "when I tug on this part, how much does this other part move", it's called a **derivative**. (SCARY!!!) Seriously, don't be scared of the word. It's just a word... and all it really means is "how much does rod X move when i tug on rod Y."

In the case of the rods above, the derivative for "how much does blue move when I tug on red" is 2. Just 2. Why is it 2? Well, that's the *multiplicative* relationship determined by the formula.

```
                                derivative
red_length = blue_length * 2
```

Notice that we always have the derivative *between two variables*. We're always looking to know how one variable moves when we change another one! If the derivative is *positive* then when we change one variable, the other will move in the *same* direction! If the derivative is *negative* then when we change one variable, the other will move in the *opposite* direction.

Consider a few examples. Since the derivative of red_length compared to blue_length is 2, then both numbers move in the same direction! More specifically, red will move *twice as much* as blue in the same direction. If the derivative had been -1, then red would move in the *opposite* direction by the same amount. Thus, given a function, the derivative represents the **direction** and the **amount** that one variable changes if you change the other variable. This is exactly what we were looking for!

Derivatives... take Two

Still a little unsure about them?... let's take another perspective...

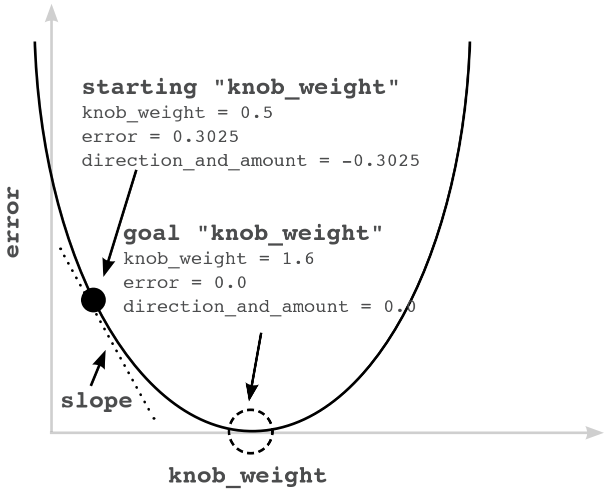
There are two ways I've heard people explain derivatives. One way is all about understanding "how one variable in a function changes when you move another variable". The other way of explaining it is "a derivative is the slope at a point on a line or curve". As it turns out, if you take a function and plot it out (draw it), the slope of the line you plot is the *same thing* as "how much one variable changes when you change the other". Let me show you by plotting our favorite function.

```
error = (goal_prediction - (input * knob_weight)) ** 2
```

Now remember... our `goal_prediction` and `input` are fixed, so we can rewrite this function:

```
error = (0.8 - (0.5 * knob_weight)) ** 2
```

Since there are only two variables left that actually change (all the rest of them are fixed), we can just take every knob_weight and compute the error that goes with it! Let's plot them on a graph!



As you can see on the right, our plot looks like a big U shaped curve! Notice that there is also a point in the middle where the error == 0! Also notice that to the right of that point, the slope of the line is *positive*, and to the left of that point, the slope of the line is *negative*. Perhaps even more interesting, the farther away from the **goal knob_weight** that you move, the *steeper* the slope gets. We like all of these properties. The *slope*'s *sign* gives us **direction** and the slope's *steepnes* gives us **amount**. We can use both of these to help find the goal knob_weight.

Even now, when I look at that curve, it's easy for me to lose track of what it represents. It's actually kindof like our "hot and cold" method for learning. If we just tried *every possible value* for knob_weight, and plotted it out, we'd get this curve. And what's really remarkable about derivatives is that they can see past our *big scary formula* for computing `error` at the top of this page and see this curve! We can actually compute the **slope** (i.e. derivative) of the line for any value of knob_weight. We can then use this slope (derivative) to figure out which **direction** reduces our error! Even better, based on the *steepness* we can get at least some idea for how far away we are (although not an exact one... as we'll learn more about later).

What you really need to know...

With derivatives... we can pick any two variables... in any formula... and know how they interact.

Take a look at this *big scary function*.

```
y = (((beta * gamma) ** 2) + (epsilon + 22 - x)) ** (1/2)
```

Here's what you need to know about derivatives. For ANY function (even this whopper) you can pick ANY TWO VARIABLES and understand their relationship with each other. For ANY function, you can pick two variables and plot them on an x-y graph like we did on the last page. For ANY function, you can pick two variables and compute how much one changes when you change the other. Thus, for ANY function, we can learn how to change one variable so that we can move another variable.

Bottom Line: In this book we're going to build neural networks. A neural network is really just one thing... a bunch of **weights** which we use to compute an **error** function. And for ANY error function (no matter how complicated), we can compute the relationship between ANY weight and the final error of the network. With this information, we can change each weight in our neural network to reduce our error down to 0... and that's exactly what we're going to do.

What you don't really need to know...

....Calculus....

So, it turns out that learning all of the methods for taking any two variables in any function and computing their relationship... takes about 3 semesters of college. Truth be told... if you went through all three semesters so that you could learn how to do Deep Learning... you'd only actually find yourself USING a very small subset of what you learned. And really... Calculus is just about memorizing and practicing every possible derivative rule for every possible function.

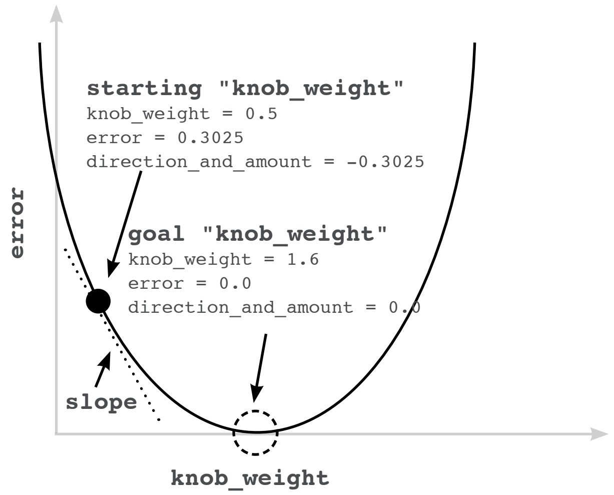
So, in this book I'm going to do what I typically do in real life (cuz i'm lazy?... i mean... efficient?) ... just look up the derivative in a reference table. All you really *need to know* is what the derivative *represents*. It's the relationship between two variables in a function... so that you can know how much one changes when you change the other. It's just the sensitivity between two variables. I know that was a lot of talking to just say "It's the sensitivity between two variables"... but it is. Note that this can include both "positive" sensitity (when variables move together) and "negative" sensitivity (when they move in opposite directions) or "zero" sensitivity... where one stays fixed regardless of what you do to the other. For example, y = 0 * x. Move x... y is always 0.

How to use a derivative to learn

Ladies and Gentlemen... Gradient Descent

What is the difference between the error and the derivative of our error and knob_weight? Well the error is just a *measure* of how much we missed. The derivative defines the *relationship* between each weight and how much we missed. In other words, it tells how *how much changing a weight contributed to the error*. So, now that we know this, how do we use it to move the error in a particular direction?

So, we've learned the relationship between two variables in a function... how do we exploit that relationship? As it turns out, this is incredibly visual and intuitive. Check out our error curve again. The black dot is where our "knob_weight" starts out at (0.5). The dotted circle is where we want it to go... our goal "knob_weight". Do you see the dotted line attached to our black dot. That's our *slope* otherwise known as our *derivative*. It tell us *at that point in the curve* how much the error changes when we change the knob_weight. Notice that it's pointed downward! It's a negative slope!



The slope of a line or curve *always points* in the opposite direction to the *lowest point* of the line or curve. So, if you have a negative slope, you *increase* your knob_weight to find the minimum of the error. Check it out!

So, how do we use our *derivative* to find the error minimum (lowest point in the error graph)? We just move in the opposite direction of the slope! We move in the opposite direction of the derivative! So, we can take each weight, calculate the derivative of that weight with respect to the error (so we're comparing two variables there... the weight and the error) and then change the weight in the *opposite* direction of that slope! That will move us to the minimum!

Let's remember back to our goal again. We are trying to figure out the **direction** and the **amount** to change our weight so that our error goes down. A derivative gives us the relationship between any two variables in a function. We use the derivative to determine the relationship between any *weight* and the *error*. We then move our weight in the *opposite* direction of the derivative to find the lowest weight. Wallah! Our neural network learns!

This method for learning (finding error minimums) is called **Gradient Descent**. This name should seem intuitive! We move in the knob_weight value *opposite the gradient* value, which *descends* our error to 0. By *opposite,* I simply mean that we increase our knob_weight when we have a negative gradient and vise versa. Visualize it in the picture above! It's like gravity!

Where is our derivative in the code?

Let's revisit our implementation to see our derivative at work

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5


for iteration in range(20):
    prediction = input * knob_weight
    error = (goal_prediction - prediction) ** 2
    derivative = input * (prediction - goal_prediction)
    knob_weight = knob_weight - derivative

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

When I run this, I see the following output.

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
                    ...

Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

So, what did we do? We computed our derivative relationship between our weight and our error, and then we *subtracted* our weight by that amount. That *subtraction* effectively means "move in the opposite direction of". If our derivative is positive, move negative. If our derivative is negative, move positive. If it is zero, don't move (you're done learning!).

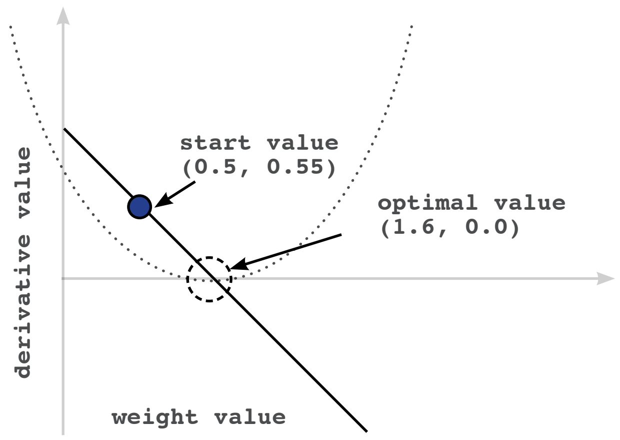
Now, when I look at code like this I find myself forgetting my previous graphs in an instant...which makes it hard to relate them! So, let's walk through all the steps at a high level again just to make sure we've got them! First, our neural network is really just a bunch of **weights** and an **error** function. Our goal is to move the weights (in this case just one) so that our error goes to zero! How do we do this? Well, it turns out we can take the derivative between two variables in *any function*. When we do this, we learn how one changes when we change the other. We can then use this to modify each weight in the **direction** that leads us to the lowest error. How do we know this direction, it is *always* in the opposite of the value of our derivative.

Many people call this method "hill climbing" because they view the error function bowl on the previous page like a little valley. Your goal is to just "follow the slope" so that you get to the bottom of the valley! This is a reasonable analogy to me, but it leaves out so much meat! What does it mean to "follow the slope downward"? What is the slope? The slope is just the relationship between a weight and the error function! Following it just means moving our weight downward when the error derivative is positive and vise versa! This collection of 4 of 5 facts in these last 2 paragraphs are key. Don't move on until you can recite them to yourself or a friend!

Learning Method: Gradient Descent

Another perspective on the same method

I want to show you a picture that describes the weight update problem from the previous page. See below.



If you tried every possible weight value and wrote down the corresponding error (given the single input/output pair), it would look like the dotted "U" shape above. If you then computed the *slope* (the derivative) of that line at every point, it would look like the straight black line above! Notice that there is a single, optimal weight at which point both the error *and* the derivative are zero. If the weight is any higher than that point, the error is positive (indicating that our prediction is too high). If the weight is any lower than that point, the error is negative (indicating that our prediction is too low). We then move our knob_weight by subtracting the derivative value.

Now, instead of focusing on all the theory of derivatives, I want to just think about what we're doing *intuitively*. Let's just try to see it plain as day. So, at the end of the day, how are we updating our weight?

```
knob_weight = knob_weight - (input * (prediction - goal_prediction))
```

So, the first thing we do is subtract our prediction and goal_prediction. At this step, we've already done most of the work! If our prediction was too low, this number will be negative. If it was too high, this number will be positive. So, at the end of the day, we're really just changing our knob_weight by *negative* the difference between our prediction and goal_prediction. In this way, if we predict too low, we'll increase our knob_weight to predict *higher* next time. If we predict too high, we'll decrease our knob_weight so that we predict *lower* next time. One more detail... we multiply by "input". Why do we do this? Well, if input happened to be 0, then there would be *no relationship* between our weight and the error. Furthermore, if our input was negative, then we need our weights to move in the opposite direction... why?... because if the input is negative moving our weight up and down has the opposite effect of when our input is positive. So, in short, our relationship between knob_weight and error is controlled by 3 variables, input, prediction, and goal_prediction and so we use all three to define that relationship.

Breaking Gradient Descent

Just Give Me The Code

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5

for iteration in range(20):
    prediction = input * knob_weight
    error = (goal_prediction - prediction) ** 2
    derivative = input * (prediction - goal_prediction)
    knob_weight = knob_weight - derivative

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

When I run this code, I see the following output...

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
                    ...

Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

Now that it works... let's break it! Play around with the starting **`weight`**, **`goal_prediction`**, and **`input`** numbers. You can set them all to just about anything and the neural network will figure out how to predict the output given the input using the weight. See if you can find some combinations that the neural network cannot predict! I find that trying to break something is a great way to learn about it.

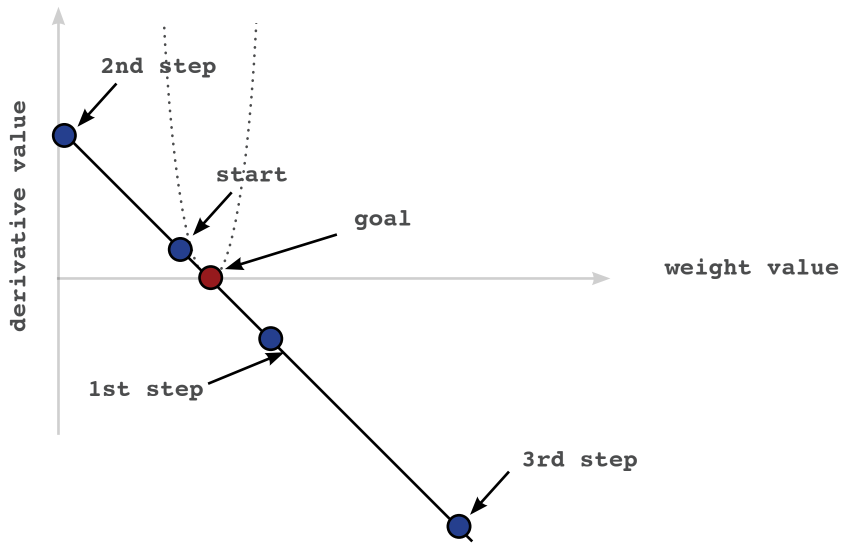
Let's try setting **`input`** to be equal to 2, but still try to get the algorithm to predict 0.8. What happens? Well, take a look at the output.

```
Error:0.04 Prediction:1.0
Error:0.36 Prediction:0.2
Error:3.24 Prediction:2.6

           ...

Error:6.67087267987e+14 Prediction:-25828031.8
Error:6.00378541188e+15 Prediction:77484098.6
Error:5.40340687069e+16 Prediction:-232452292.6
```

Woah! That's not what we wanted! Our predictions exploded! They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step! In other words, every update to our weight **overcorrects**! In the next section, we'll learn more about how to combat this phenomenon.

Divergence

Sometimes... neural networks explode in value... oops?



So what really happened? The explosion in error on the previous page is caused by the fact that we made the input larger. Consider our update rule.

```
knob_weight = knob_weight - (input * (prediction - goal_prediction))
```

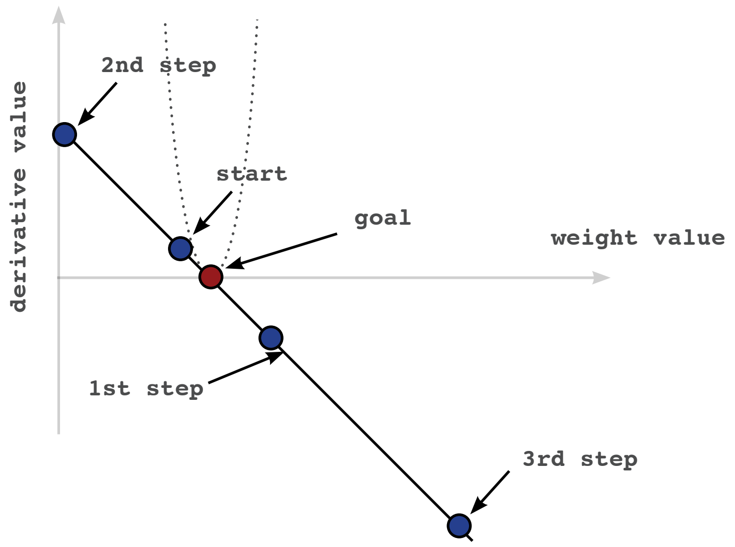
If our input is sufficiently large, this can make our weight update large even when our error is small. What happens when you have a large weight update and a small error? It overcorrects!!! If the new error is even bigger, it overcorrects even more!!! This causes the phenomenon that we saw on the previous page, called **divergence**.

You see, if we have a BIG input, then the prediction is VERY sensitive to changes in the weight (since prediction = input * knob_weight). This can cause our network to overcorrect. In other words, even though our knob_weight is still only starting at 0.5, our derivative at that point is *very steep*. See how tight the u shaped error curve is in the graph above?

This is actually really intuitive. How do we predict? Well, we predict by *multiplying* our input by our knob_weight. So, if our input is *huge*, then small changes in our knob_weight are going to cause BIG changes in our prediction!! The error is REALLY *sensitive* to our knob_weight. Aka... the derivative is really big! So, how do we make it smaller?

Introducing.... Alpha

The simplest way to prevent overcorrecting our weight updates.



So, what was the problem we're trying to solve? The problem is this: if the input is too big, then our weight update can overcorrect. What is the symptom? The symptom is that when we overcorrect, our new derivative is even *larger in **magnitude*** than when we started (although the sign will be the opposite). Stop and consider this for a second. Look at the graph above to understand the symptom. The 2nd step is even farther away from the goal... which means the *derivative* is even greater in magnitude! This causes the 3rd step to be even farther away from the goal than the second step, and the neural network continues like this, demonstrating **divergence**.

The symptom is this overshooting. The solution is to *multiply the weight update by a fraction* to make it smaller. In most cases, this involves multiplying our weight update by a single real-valued number between 0 and 1, known as **alpha**. One might note, this has no affect on the *core issue* which is that our input is larger. It will also reduce the weight updates for inputs that aren't too large. In fact, finding the appropriate alpha, even for state-of-the-art neural networks, is often done simply by guessing. You watch your error rates. If they start diverging (going up), then your alpha is too high, and you decrease it. If learning is happening too slowly, then your alpha is too low, and you increase it. There are other methods than simple gradient descent that attempt to counter for this, but gradient descent is still very popular.

Alpha In Code

Where does our "alpha" parameter come in to play?
So we just learned that **alpha** reduces our weight update so that it doesn't overshoot. How does this affect our code? Well, we were updating our weights according to the following formula.

```
knob_weight = knob_weight - derivative
```

Accounting for alpha is a rather small change, pictured below. Notice that if alpha is small (say...0.01), it will reduce our weight update considerably, thus preventing it from overshooting.

```
knob_weight = knob_weight - (alpha * derivative)
```

Well, that was easy! So, let's install alpha into our tiny implementation from the beginning of this chapter and run it where input = 2 (which previously didn't work)

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5
alpha = 0.1

for iteration in range(20):
    prediction = input * knob_weight
    error = (goal_prediction - prediction) ** 2
    derivative = input * (prediction - goal_prediction)
    knob_weight = knob_weight - (alpha * derivative)

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

```
Error:0.04 Prediction:1.0
Error:0.0144 Prediction:0.92
Error:0.005184 Prediction:0.872

            ...

Error:1.14604719983e-09 Prediction:0.800033853319
Error:4.12576991939e-10 Prediction:0.800020311991
Error:1.48527717099e-10 Prediction:0.800012187195
```

```
What happens
when you make
alpha crazy
small or big?
What about
making it
negative?
```

Wallah! Our tiniest neural network can now make good predictions again! How did I know to set alpha to 0.1? Well, to be honest, I just tried it and it worked. And despite all the crazy advancements of deep learning in the past few years, most people just try several orders of magnitude of alpha (10,1,0.1,0.01,0.001,0.0001) and then tweak from there to see what works best. It's more art than science. There are more advanced ways which we can get to later, but for now, just try various alphas until you get one that seems to work pretty well. Play with it!

Memorizing

Ok... it's time to really learn this stuff
This may sound like something that's a bit intense, but I can't stress enough the value I have found from this exercise. The code on the previous page, see if you can build it in an iPython notebook (or a .py file if you must) from *memory*. I know that might seem like overkill, but I (personally) didn't have my *click* moment with neural networks until I was able to perform this task.

Why does this work? Well, for starters, the only way to *know* that you have gleaned all the information necessary from this chapter is to try to produce it just from your head. Neural networks have lots of small moving parts, and it's easy to miss one.

Why is this important for the rest of the chapters? In the following chapters, I will be referring to the concepts discussed in this chapter at a *faster pace* so that I can spend pleny of time on the newer material. It is *vitally important* that when I say something like "add your alpha parameterization to the weight update" that it is at least immediately apparent to which concepts from this chapter I'm referring.

All that is to say, memorizing small bits of neural network code has been hugely beneficial for me personally, as well as to many individuals who have taken my advice on this subject in the past.

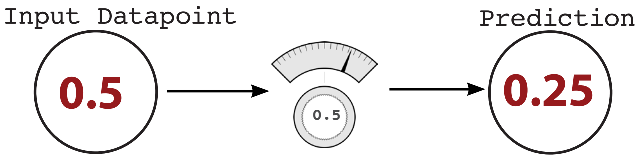
Neural Networks with Multiple Inputs 4
Leveraging Matrix Algebra

IN THIS CHAPTER

Expanding to Multiple Input Values

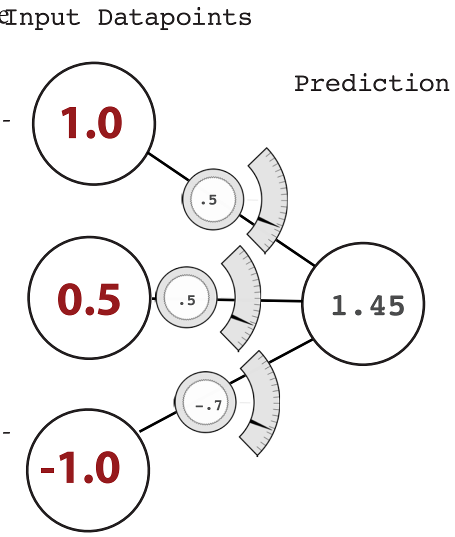
Our previous chapter took one number to predict... now we'll use 3

In the previous chapter, we learned about what a neural network is made of and how it learns. A neural network is really just a collection of weights that are part of an error function. We then look for the derivative between each weight and the error function and use this to modify our weights until the error reaches 0. In the previous chapter, we made a neural network with a single input, single weight, and single output.



In this chapter, we're going to expand to a neural network with 3 inputs. We will find that this is pretty much the same as when we have a single input... we just do all the operations we did for a single input in parallel on all three weights at the same time. Let's check it out!

As a small challenge for you, see if you can take the code from the previous chapter and modify it to have three inputs. What challenges do you face?
-------------STOP AND ATTEMPT----------------------
Perhaps the first challenge is "how do you predict with three inputs and one output?" Well, before we just multiplied our input by our weight. Do that for each of your weights and then sum their respective outputs into a single number. That's your prediction! Now, see if you can get it to learn the example on the right here. with 1, 0.5, and -1 as inputs and 1.45 as the predicted output.
-------------STOP AND ATTEMPT----------------------
Hopefully you were able to successfully train three weights to predict 1.45 given those three inputs. Great work! In this chapter, we're also going to make another intersting extension of our work. All of our neural networks so far have learned a single "training example". In other words, it has taken one set of inputs and generated a single output. In this chapter, we're also going to learn how to train a neural network that can learn multiple training examples... entire datasets in fact. These training datasets will start to show us how we can leverage deep learning to make predictions in the real world. And to that end, we're going to spend a few minutes talking about what it means to import patterns from the real world into our computers in the form of "datasets".



What We're Going to Build

At the end of this chapter, we'll build and understand the following code...

I don't expect everything to be made clear quite yet, but given a solid understanding of the previous chapter, you should be able to generally follow along with what's going on in these few lines of code. Don't worry if you have some questions... you should. However, I find that it's always a good idea to know "where we're going" as a frame of reference.

```
import numpy as np

weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [[ 1, 0, 1 ],
                          [ 0, 1, 1 ],
                          [ 0, 0, 1 ],
                          [ 1, 1, 1 ],
                          [ 0, 1, 1 ],
                          [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights # [1,0,1]
goal_prediction = walk_vs_stop # equals 0... i.e. "stop"

for iteration in range(100):
    prediction = input.dot(weights)
    error = np.sum((prediction - goal_prediction) ** 2)
    delta = prediction - goal_prediction
    weights -= (alpha * (input.T.dot(delta)))
    print "Average Error:" + str(error) + " Prediction:" + str(prediction)
```

And when I run this code I see...

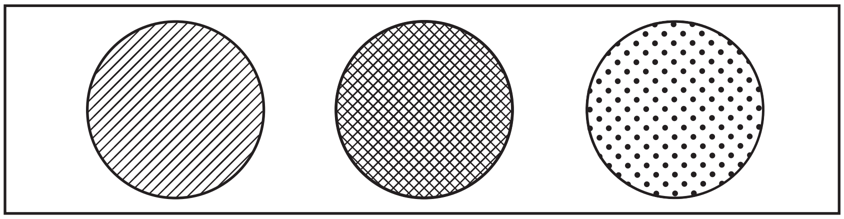
```
Average Error:4.0652 Prediction:[-0.2  -0.22 -0.7   0.28 -0.22 -0.2 ]
Average Error:0.778488 Prediction:[ 0.34  0.52 -0.27  1.13  0.52  0.34]
Average Error:0.51437472 Prediction:[ 0.3   0.65 -0.23  1.18  0.65  0.3 ]
...
Average Error:3.00651151819e-07 Prediction:[-0.  1. -0.  1.  1. -0.]
Average Error:2.63337305604e-07 Prediction:[-0.  1. -0.  1.  1. -0.]
Average Error:2.30654484785e-07 Prediction:[-0.  1. -0.  1.  1. -0.]
```

There's a few new things here that we're going to learn about in this chapter. The first is a new variable, `delta`. We'll also learn about how to use matrices (the "tables of numbers above), and matrix functions like "dot". However, on the hole, even though we have some new "tools" for learning in this neural network, we really are just doing the same thing we did for single-weight neural network with our 3-weight neural network. These tools just make it easier. Let's jump right in!

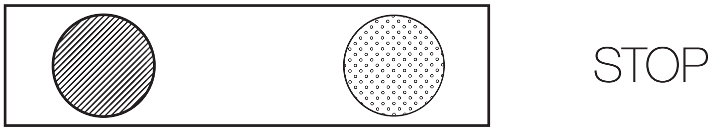
The Street Light Problem

A toy problem for our first neural network to solve.

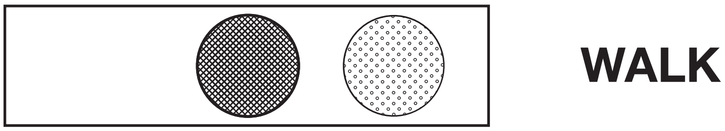
Consider yourself approaching a street corner in a foreign country. As you approach, you look up and realize that the street light is quite unfamiliar. How can you know when it is safe to cross the street?



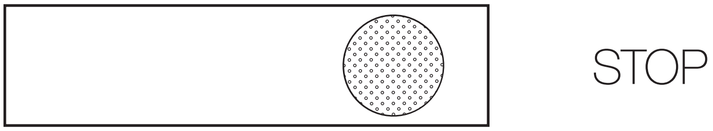
You can know when it is safe to cross the street by interpreting the streetlight. However, in this case, we don't know how to interpret it! Which "light" combinations indicate when it is time to walk? Which indicate when it is time to stop? To solve this problem, you might sit at the street corner for a few minutes observing correlation between each light combination and whether or not people around you choose to walk or stop. You take a seat and record the following pattern.



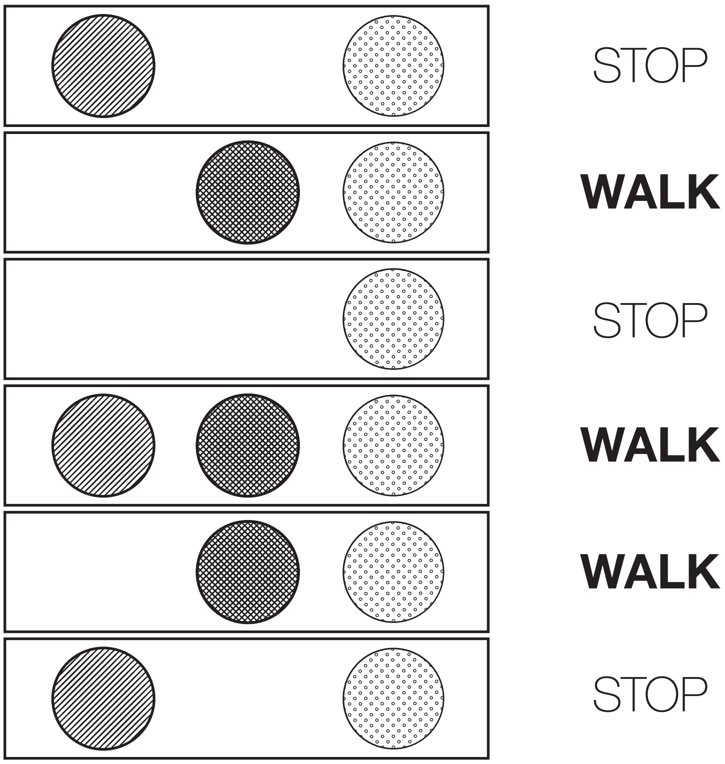
Ok, nobody walked at the first light. At this point your thinking, "man, this pattern could be anything. The left light or the right light could be correlated with stopping, or the central light could be correlated with walking." There's no way to know. Let's take another datapoint.



People walked! Ok, so something changed with this light that changed the signal. The only thing we know for sure is that the far right light doesn't seem to indicate one way or another. Perhaps it is irrelevant. Let's collect another datapoint.



Now we're getting somewhere! Only the middle light changed this time, and we got the opposite pattern. Our working hypothesys is that the middle light indicates when people feel safe to walk. Over the next few minutes, we recorded the following six light patterns, noting when people seemed to walk or stop. Do you notice a pattern overall?



As hypothesized on the previous page, there is a perfect correlation between the middle (criss-cross) light and whether or not it is safe to walk. You were able to learn this pattern by observing all of the individual datapoints and searching for correlation. This is what we're going to train our neural network to do.

Preparing our Data

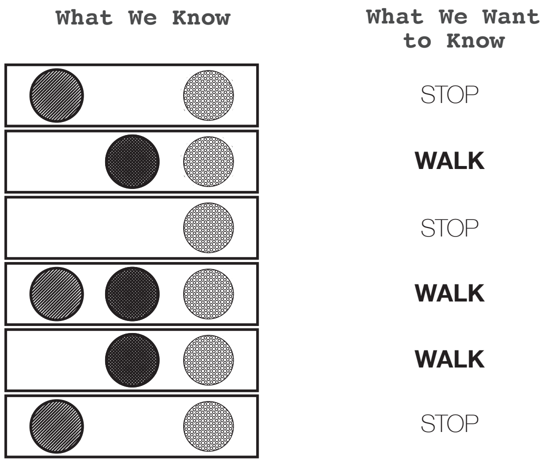
Neural Networks Don't Read Streetlight

In the previous chapters, we learned about supervised algorithms. We learned that they can take one dataset and turn it into another. More importantly, they can take a dataset of what we know and turn it into a dataset of what we want to know.

So, how do we train a supervised neural network? Well, we present it with two datasets and ask it to learn how to transform one into the other. Think back to our streetlight problem. Can you identify two datasets? Which one do we always know? Which one do we want to know?

We do indeed have two datasets. On the one hand, we have six streetlight states. On the other hand, we have 6 observations of whether people walked or not. These are our two datasets.

So, we can train our neural network to convert from the dataset we know to the dataset that we want to know. In this particular "real world example", we know the state of the streetlight at any given time, and we want to know whether it is safe to cross the street.

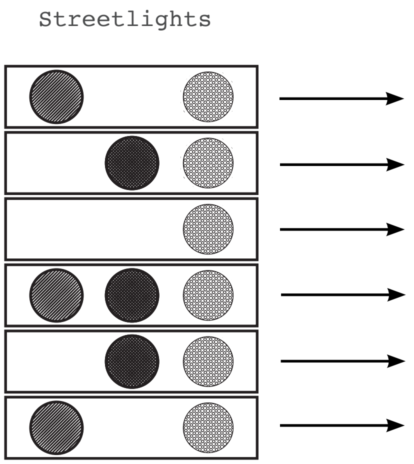


So, in order to prepare this data for our neural network, we need to first split it into these two groups (what we know and what we want to know). Note that we could attempt to go backwards if we swapped which dataset was in which group. For some problems, this works.

Matrices and the Matrix Relationship

Translating your streetlight into math.

Math doesn't understand streetlights. As mentioned in the previous section, we want to teach our neural network to translate a streetlight pattern into the correct stop/walk pattern. The operative word here is pattern. What we really want to do is mimick the pattern of our streetlight in the form of numbers. Let me show you what I mean.



Streetlight **Pattern**

$$\begin{array}{ccc}
1 & 0 & 1 \\
0 & 1 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1
\end{array}$$

Notice in the matrix on the right that we have mimicked the pattern from our streetlights in the form of 1s and 0s. Notice that each of the lights gets a column (3 columns total since there are three lights). Notice also that there are 6 rows representing the 6 different streetlights that we observed.

This structure of 1s and 0s is called a matrix. Furthermore, this relationship between the rows and columns is very common in matrices, especially matrices of data (like our streetlights).

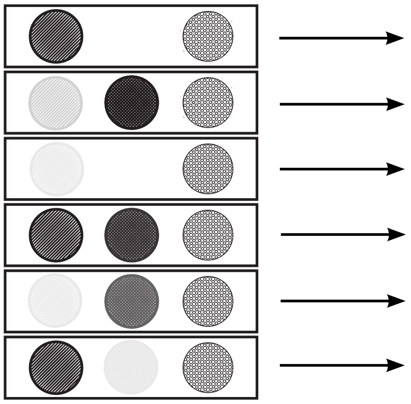
In data matrices, it is convention to give each recorded example a single row. It is also convention to give each thing being recorded a single column. This makes it easy to read.

So, a column contains every state we recorded a thing in. In this case, a column contains every on/off state we recorded of a particular light. Each row contains the simultaneous state of every light at a particular moment in time. Again, this is common.

Good data matrices perfectly mimick the outside world.

Our data matrix doesn't have to be all 1s and 0s. What if the streetlights were on "dimmers" and they turned on and off at varying degrees of intensity. Perhaps our streetlight matrix would look more like this.

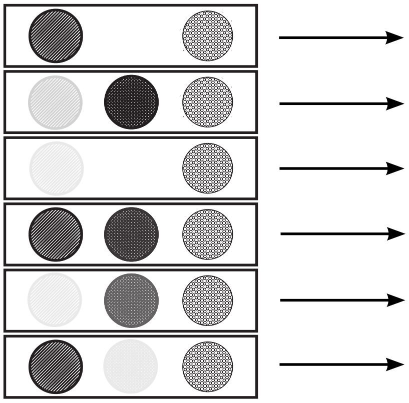
`Streetlights` `Streetlight` **`Matrix A`**



$$\begin{array}{ccc} .9 & .0 & 1 \\ .2 & .8 & 1 \\ .1 & .0 & 1 \\ .8 & .9 & 1 \\ .1 & .7 & 1 \\ .9 & .1 & 0 \end{array}$$

Matrix A above is a perfectly valid matrix. It is mimicking the patterns that exist in the real world (streetlight) so that we can ask our computer to interpret them. Would the following matrix still be valid?

`Streetlights` `Streetlight` **`Matrix B`**



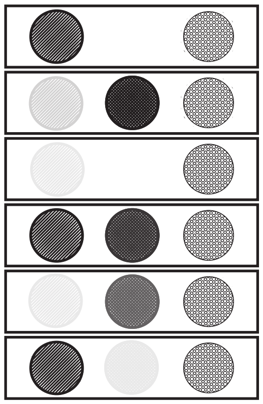
$$\begin{array}{ccc} 9 & 0 & 10 \\ 2 & 8 & 10 \\ 1 & 0 & 10 \\ 8 & 9 & 10 \\ 1 & 7 & 10 \\ 9 & 1 & 0 \end{array}$$

In fact, this matrix (B) is still valid. It adquality captures the relationships between various training examples (rows) and lights (columns). Note that "Matrix A" * 10 == "Matrix B" (A * 10 == B). This actually means that these matrices are scalar multiples of each other.

Matrix A and B both contain the same underlying pattern.

The important takeaway here is that there are an infinite number of matrices that perfectly reflect the streetlight patterns in our dataset. Even the one below is still perfect.

`Streetlights` `Streetlight` **`Matrix C`**



18	0	20
4	16	20
2	0	20
16	18	20
2	14	20
18	2	0

It's important to recognize that "the underlying pattern" is not the same as "the matrix". It's a property of the matrix. In fact, it's a property of all three of these matrices (A, B, and C). The pattern is what each of these matrices is expressing. The pattern also existed in the streetlights. This input data pattern is what we want our neural network to learn to transform into the output data pattern.

However, in order to learn the output data pattern, we also need to capture the pattern in the form of a matrix. Let's do that below.

Note that we could reverse the 1s and 0s here and the output matrix would still be capturing the underlying STOP/WALK pattern that's present in our data. We know this because regardless of whether we assign a 1 to WALK or to STOP, we can still de-code the 1s and 0s into the underlying STOP/WALK pattern. We call this resulting matrix a lossless represen-tation because we can can perfectly convert back and forth between our stop/walk notes and the matrix.

STOP	0
WALK	1
STOP	0
WALK	1
WALK	1
STOP	0

Note that this matrix only has one column because we're only representing one "thing" (whether people stopped or walked)

Creating a Matrix or Two in Python

Importing our matrices into Python

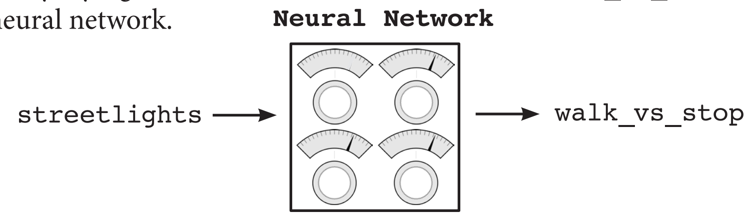
So, we've converted our streetlight pattern into a matrix (the one with just 1s and 0s). Now, we want to create that matrix (and more importantly, it's underlying pattern) in Python so that our neural network can read it. Python has a special library built just for handling matrices called numpy. Let's see it in action.

```
import numpy as np
streetlights = np.array( [ 1, 0, 1 ],
                         [ 0, 1, 1 ],
                         [ 0, 0, 1 ],
                         [ 1, 1, 1 ],
                         [ 0, 1, 1 ],
                         [ 1, 0, 1 ] ] )
```

If you're a regular Python user, something should be very striking from this code. A matrix is really just a list of lists! It's an array of arrays! What is numpy? Numpy is really just a fancy wrapper for an array of arrays that gives us special matrix-oriented functions. Let's create a numpy matrix for our output data too.

```
walk_vs_stop = np.array( [ 0 ],
                         [ 1 ],
                         [ 0 ],
                         [ 1 ],
                         [ 1 ],
                         [ 0 ] ] )
```

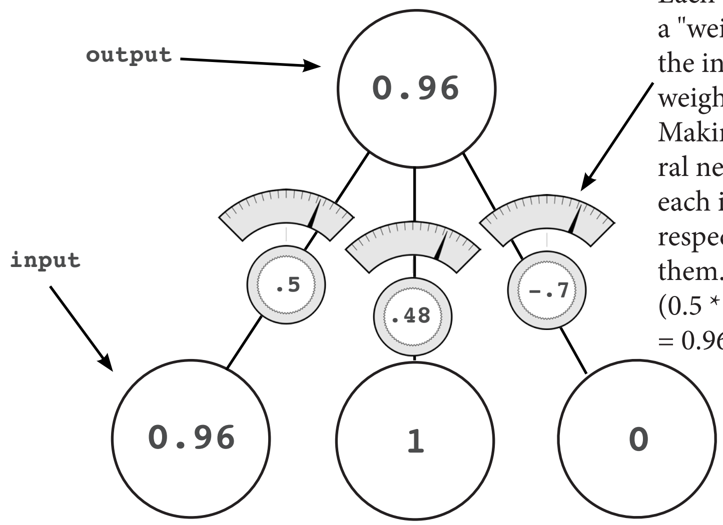
So, what will we want our neural network to do? Well, we will want it to take our `streetlights` matrix and learn to transform it into our `walk_vs_stop` matrix. More importantly, we will want our neural network to take any matrix containing the same underlying pattern as `streetlights` and transform it into a matrix that contains the underlying pattern of `walk_vs_stop`. More on that later. Let's start by trying to transform `streetlights` into `walk_vs_stop` using a neural network.



What is a Neural Network

A neural network is really just a bunch of matrices.

A neural network, in its simplest form, is really just a bunch of weights, organized into matrices (just like our data). The more complex the network, the more matrices it has. In our case, we're going to start off with a "single matrix" neural network. Let's see what it looks like.



Each of these "knobs" is called a "weight" because it "weights" the input. We store all 3 of these weights in the weight matrix. Making a prediction with our neural network is a simple as taking each input, multiplying it by its respective weight, and summing them. Try it!
(0.5 * 0.96) + (0.48 * 1) + (-0.7 * 0) = 0.96

The neural network above has three inputs and generates one output. In the middle, we have three "knobs" or "weights". When given one input example, the neural network mutliples each of the three inputs by its three weights and then sums them. The result is the output of the neural network. Check it out:

(0.96 * 0.5) + (1 * 0.48) + (0 * -0.7) = 0.96

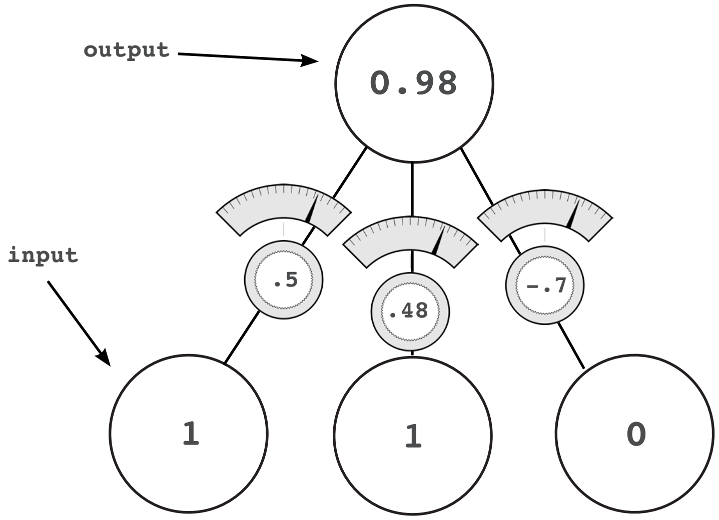
This should look very familiar! It's basically like having 3 simultaneous networks from the previous chapter! Given any three inputs, this tiny neural network can generate an output. This output might not be the output we want, but it will generate one. Notice that the output the neural network creates is entirely dependent on the positions of our "knobs" (the values of our weights). Learning is accomplished by finding the correct set of weights that can transform our input data to our output data.

Something to highlight... in order to train this network... we'll adjust each weight in the correct direction by the correct amount so that when the weights predict together, they predict the output data perfectly. Sound familiar?

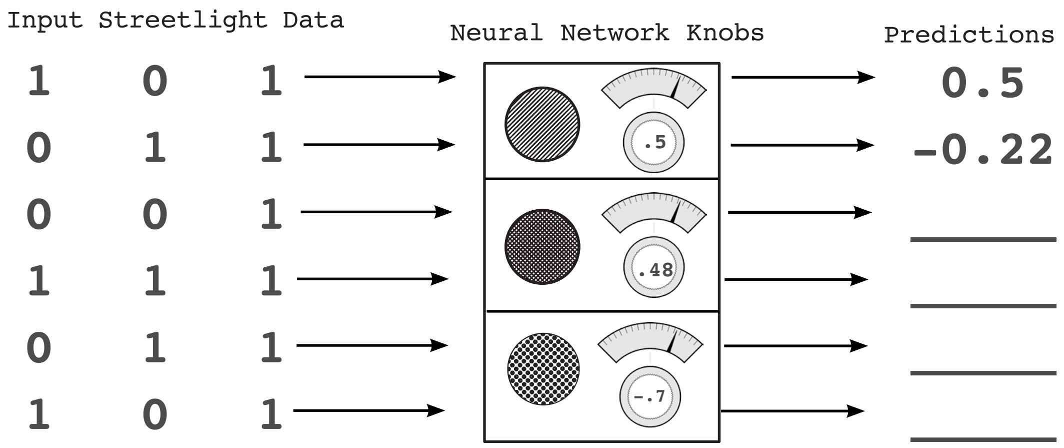
How Does A Neural Network Predict?

This neural network computes a "weighted sum" of the inputs.

In the previous chapter, our "world's smallest neural network" predicted by multiplying an input value by a weight. Now that we have 3 input values and 3 weights, we do the same for each! (and just sum them at the end). In this case, it makes a prediction (0.98) by computing a weighted sum of three inputs (in this case, 1, 1, and 0 are our three inputs). This predictions means that... given the state of our three lights, there is a 98% chance that it is safe to cross the street (according to our network). This idea should feel very comfortable to you: look at the network below... it really is just like having 3 networks from the previous chapter... side by side... where we sum their respective predictions.



Try to fill in the last 4 predictions below!



How Does A Neural Network Learn?

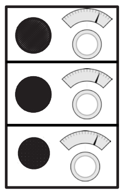
It learns just like the "single weight" network from the previous chapter... it just does the same operation on all 3 weights in parallel.

So, if our neural network is really just a weighted sum, how does it learn? Remember how the network from the previous chapter learned? We calculated the relationship between our error variable and the weight variable... we do the same here... just to all three weights. Predict... calculate the derivative for each weight... modify each weight... repeat

Step 1: Make a Prediction

We did this on the last page, with the following results. Our simple neural network is just a weighted sum of the inputs. We take each input, multiply it by its respective weight, and sum the outputs. We then use this prediction to calculate our error.

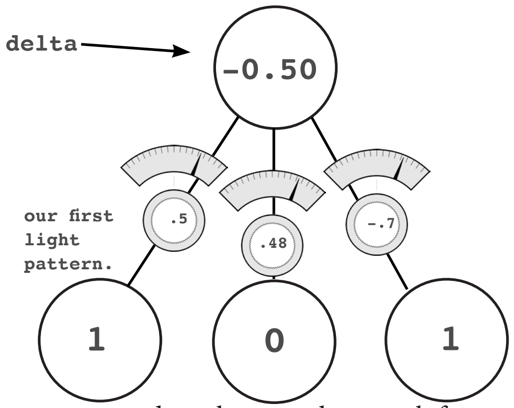
Streetlights			Predictions
1	0	1	0.5
0	1	1	-0.22
0	0	1	-0.7
1	1	1	0.28
0	1	1	-0.22
1	0	1	0.5



Step 2: Compare to Truth Pattern

Just like our previous neural network, we are going to calculate the derivative of each weight with respect to the error. This process has two steps: 1) calculate prediction - goal_prediction and 2) multiply that number by our input. Now that we have three weights, we actually have a different input for each weight. However, that first step (the difference) is the same for all of them... so we can just calculate it once and call it `delta`.

(Predictions - Truth)		Delta
(.50 - 0)		-0.50
(-.22 - 1)	equals	-1.22
(-.70 - 0)		-0.70
(.28 - 1)	=	-0.72
(-.22 - 1)		-1.22
(.50 - 0)		-0.50



Step 3: Learn the Pattern

What is `delta`? Well, it's the first half of our derivative calculation (`prediction - goal_prediction`). Now we can use that `delta` to calculate the derivative for each weight. For example, we take the left input and multiply it by the `delta` to calcualte the derivative for our left weight. We then repeat this process for the center and right weights... giving us all three derivatives! Notice that this is the exact same process we used on the neural network from the previous chapter... we're just doing this operation for each weight in our neural network.

Slow down! What is "delta"?

Let's pretend we were actually training 3 single-weight matrices...

Let's review how we updated our knob_weight in the previous chapter. Here's what we did!

```
derivative = (input * (prediction - goal_prediction))
knob_weight = knob_weight - derivative
```

So, if we were to compute this for each of our knob_weights in our 3-node network...

```
left_derivative = (left_input * (prediction - goal_prediction))
left_knob_weight = left_knob_weight - left_derivative

middle_derivative = (middle_input * (prediction - goal_prediction))
middle_knob_weight = middle_knob_weight - middle_derivative

right_derivative = (right_input * (prediction - goal_prediction))
right_knob_weight = right_knob_weight - right_derivative
```

Here's the key lesson... each of our 3 weights contribute to the prediction... and at the end of the day, if the prediction is...say... too high... then all of our weights need to predict a little bit lower. Each weight's relationship with the error isn't any different than it was in the previous chapter...we just have more weights... so we have to repeat the process of calculating the derivative for each weight.

Now, a part of this process is exactly the same for every weight derivative calculation. The part that's the same is the (prediction - goal_prediction) part. So, it saves time to just precompute that part and put it in a variable... so we precompute it and call it delta.

```
delta = prediction - goal_prediction

left_derivative = (left_input * delta)
left_knob_weight = left_knob_weight - left_derivative

middle_derivative = (middle_input * delta)
middle_knob_weight = middle_knob_weight - middle_derivative

right_derivative = (right_input * delta)
right_knob_weight = right_knob_weight - right_derivative
```

See how it saves time? We don't have to calculate (prediction - goal_prediction) for every weight we want to update! It's convenient!

Surprise! There's two steps in calculating the derivative! The first is calculating delta and the second is multiplying that delta by the input (which gives us the derivative). Why multiply by input? Well, it gives us two desirable behaviors.

Property 1: if the input is zero, the weight update (derivative) is also zero. If the input is zero, then the weight is irrelevant to the prediction... so modifying it would be silly!

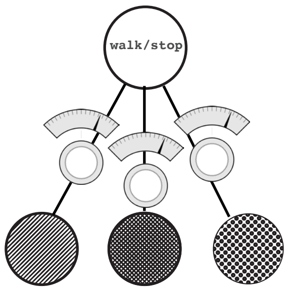
Property 2: if the input is negative, then increasing the weight actually decreases the prediction. So, we need our weight update to change directions somehow to account for negative input. Multiplying by our input also has the affect of reversing the direction of the update.

Building our Neural Network in Python

How a Neural Network Learns in Python

The neural network is really just those three weights. We store them in a matrix. Check it out!

```
import numpy as np
weights = np.array([ 0.5, 0.48, -0.7 ])
```



Notice something. We have 3 weights. Why do we have three weights? We have three inputs. Notice another something. We have one column. Why do we have one column? We only have one output! If we had another output, we'd have another column of weights connecting the input to that second output. Neat!
Ok, time for some shorthand. Let's say we store our inputs in another array like so. `input = streetlights[0] # [ 1, 0, 1]`

What do we want to do to predict? Well, we want to take each input and multiply it by its respective weight. Then, we want to sum the output of these multiplications. In short, it's a weighted sum. Well, when you store your weights in a matrix, and your input in a matrix, and you want to do a weighted sum... there's a really nice `numpy` function to help you! It's called "dot". Check it out!

```
prediction = input.dot(weights)# <-- dot product here
print prediction # prints [ -0.20 ]
```

What happened? Well, all in one line, `numpy` took each of our inputs, multiplied them by each corresponding weight, and summed the multiplications.... all in one convenient function called "dot"!

Congratulations! If you were following along with the Python code, you just made your first neural network prediction! Your neural network should have predicted `-0.20`.

You may be wondering, what does the `-0.20` mean? In this case, it's a random prediction. Don't be surprised, our neural network doesn't know anything yet. We haven't taught it our pattern! In the next section, we're going to learn how neural networks learn.

Teaching Our Neural Network in Python

How can we adjust the weights to predict more accurately?

Step 1: Make a Prediction

```
input = np.array([ 1, 0, 1 ]) # our first streetlight
prediction = input.dot(weights) # predicts -0.20
```

Step 2: Compare to Truth Pattern

Remember the next step after making our prediction? We compute our delta! Why? Well, all three of our derivatives for our weights need to know (prediction - goal_prediction), so we just compute it once to be more efficient!

```
goal_prediction = walk_vs_stop[0] # this equals 0... i.e. "stop"
delta = prediction - goal_prediction
```

Step 3: Learn the Pattern

```
weights = weights - (input * delta)
```

So there's a little magic going on behind the scenes here. It's really important to know what "input" and "delta" are. In this case, "delta" is a single number (0.20) and "input" is the vector we set it to be (1,0,1). When you multiply two numpy variables together like that... and one is a single number... and the other is a vector... it multiplies every number in the vector by the single number. So, in this case, we multiplied every input by 0.20. The resulting vector is [0.20, 0, 0.20] which we used to update each of our weights. The far left 0.20 updated the left weight, the 0 updated the middle weight (by nothing), and the right 0.20 updated the right weight... all in one line of code.

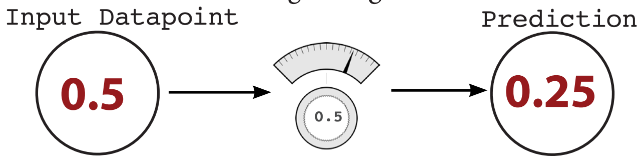
You might be wondering "why in the world did we do it like this? It's so hard to see what's going on?!?!"... and man i gotta admit... that's kinda true. The benefit will become clear later. Just trust me. For now, just see if you can "read" this code and follow along with what's happening. You'll see if you "predict" again... it will be closer to 0!

```
import numpy as np
weights = np.array([[0.5],[0.48],[-0.7]])
input = streetlights[0] # [1,0,1]
prediction = input.dot(weights)
goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop"
delta = prediction - goal_prediction
weights = weights - (input * delta)
```

Quick Review from 10,000 Feet

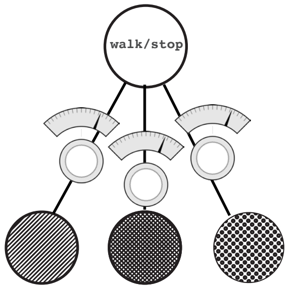
We've been very focused on new pieces... let's remember how they come together

In the previous chapter, we built a neural network with a single weight.



```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5

for iteration in range(20):
    prediction = input * knob_weight
    error = (prediction - goal_prediction) ** 2
    derivative = input * (prediction - goal_prediction)
    knob_weight = knob_weight - derivative

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

What did we learn? We learned that a neural network is two things: a weight and an error function. Learning is just a matter of moving our weight until our error goes to 0. Because our error and our weight are part of a function, we can calculate the derivative relationship between them... which lets us know how much moving the weight causes the error to change. In fact, adding the derivative to our weight would cause the error to increase and subtracting the derivative to our weight causes the error to decrease. So, if we just recompute the derivative and subtract it from our weight a few dozen times... our neural network learns to predict correctly! Wallah! Ladies and gentlemen... single weight learning.



So, now that we have three inputs, what do we know? Well, a neural network is still just two things... weights (plural) and an error function. Learning is just a matter of moving each of our weights until our error goes to 0. Because our error and our weights are a function, we can calculate the derivative relationship between the error and each one of our weights. This lets us know how changing each weight causes our error to change. In fact, for each weight...adding its derivative error causes the error to increase and subtracting its derivative causes the error to decrease. Thus, a neural network learns by repeatedly computing the derivatives for each weight... and subtracting the weight by them. Repeat this process... our error falls to 0... and our neural network learns to predict correctly! Wallah! Multi-weight, multi-input learning!

I structured the two previous paragraphs very intentionally. Notice that each of the steps is the same... in the same order. Honestly, a neural network with three inputs/weights is the same thing as a neural network with one weight... we just repeat the process for each input/weight (and cache the delta for efficiency).

Oh... and remember that thing called alpha?... we do that with 3 weights too to prevent divergence! Let's put it all together on the next page!

Putting it all together in Python (verbose version)

We've been very focused on new pieces... let's remember how they come together

In the previous chapter, we built a neural network with a single weight. In this chapter, we learned that we can use the same logic to train a 3-weight neural network. Before showing you the more "polished version" of the code, I want to show you a version of the code that should show you the similarities between the single-weight neural network and our 3-weight neural network. After that, we'll change our code (to do the same thing) but look a bit prettier.

```
import numpy as np

weights = np.array([0.5,0.48,-0.7])
left_weight = weights[0]
mid_weight = weights[1]
right_weight = weights[2]

alpha = 0.1

streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] # first streetlight [1,0,1]

left_input = input[0]
mid_input = input[1]
right_input = input[2]

goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop"

for iteration in range(20):
    prediction =  (left_input * left_weight) + \
                   (mid_input * mid_weight) + \
                   (right_input * right_weight)

    error = (prediction - goal_eeprediction) ** 2
    delta = prediction - goal_prediction

    left_derivative = left_input * delta
    mid_derivative = mid_input * delta
    right_derivative = right_input * delta

    left_weight = left_weight - (alpha * left_derivative)
    mid_weight = mid_weight - (alpha * mid_derivative)
    right_weight = right_weight - (alpha * right_derivative)

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

Putting it all together in Python (polished version)

For reference, I'm going to print the single-weight neural network again first.

```
knob_weight = 0.5
goal_prediction = 0.8
input = 0.5

for iteration in range(20):
    prediction = input * knob_weight
    error = (prediction - goal_prediction) ** 2
    derivative = input * (prediction - goal_prediction)
    knob_weight = knob_weight - derivative

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

And now, I'm going to show you the polished version of the 3-weight neural network. Note, that it gets the exact same output in every way as the "verbose version" on the previous page; it just does so with far fewer lines of code.

```
import numpy as np

weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [[ 1, 0, 1 ],
                          [ 0, 1, 1 ],
                          [ 0, 0, 1 ],
                          [ 1, 1, 1 ],
                          [ 0, 1, 1 ],
                          [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] # [1,0,1]
goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop"

for iteration in range(20):
    prediction = input.dot(weights)
    error = (goal_prediction - prediction) ** 2
    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))

    print "Error:" + str(error) + " Prediction:" + str(prediction)
```

So, I don't have space on this page to explain all the similarities, so I want you to just start by first seeing the similarities between the single-weight and the "verbose" 3-weight matrix. After those are clear in your mind, then study the "verbose" 3-weight and the "polished" 3-weight. (And no worries... we'll walk through the "polished" one more in a second.. it just helps to see them side by side).

The big takeaway here is that our use of "matrices" really makes for concise code... but they don't do anything different than we were doing before. The tough part (and what you want to learn) is to become familiar with the API so that you can "read" these matrices in just the same way you could "read" individual variables before.

The Differences Line by Line

For the lines that look different... here's how they're actually the same.

The value of each variable between our "verbose" and "polished" versions, is identical, but the exact API methods we're using are slightly different. Let's investigate those differences.

Verbose **Polished**

```
prediction = (left_input * left_weight)+\ weights=np.array([0.5,0.48,-0.7])
             (mid_input * mid_weight)+\   input = streetlights[0] # [1,0,1]
             (right_input * right_weight) prediction = input.dot(weights)
```

The difference between these two versions is the use of the "dot" function. Both just take two matrix rows (also known as "vectors") of equal length, multiply each place respectively (left with left, middle with middle, right with right), and then sum them. It's a weighted sum operation. Notice that the matrix rows (vectors) must be of equal length.

```
left_derivative = left_input * delta
mid_derivative = mid_input * delta
right_derivative = right_input * delta

left_weight = left_weight - (alpha * left_derivative)
mid_weight = mid_weight - (alpha * mid_derivative)
right_weight = right_weight - (alpha * right_derivative)

                              weights = weights - (alpha * (input * delta))
```

Now we see that the "polished" code can do in a single line what takes 6 lines for the "verbose" version. The first 3 lines of "verbose" code are accomplished via simply (input * delta). So, given that input and delta are both matrix rows (vectors) of equal length (3), if we multiply them together, the result is another matrix row (vector) of length 3, where the left value in each was multiplied together, and the same with the middle values, and right values. A close eye will also notice that if we summed the resulting vector... it'd be the same as calling "dot". Multiplying each respective value in two identically sized matrices (or vectors) is called elementwise multiplication because we multiply each element in one matrix by each corresponding element in the same position in another matrix (or row of a matrix... i.e. vector).

Remember what alpha is? It's a single number (0.1). So, we just got done multiplying two matrix rows (vectors)... what happens when we multiply a single number by a matrix row (vector)? Well, that single number gets multiplied by every number in the matrix row (vector)... resulting in an identically sized matrix row (vector). This is a funny part of the API as the star (*) means different things depending on what's being multiplied. If it's two numbers... it's basic multiplication... two matrices... it's elementwise multiplication... and a matrix and a number it' just multiplies every value in the matrix by that number...
And finally "weights = weights - " is just elementwise subtraction which is what it sounds like... subtracting each value in one matrix by each value in another according to position.

Learning the whole dataset!

So... in the last few pages... we've only been learning one streetlight. Don't we want to learn them all?

So far in this book, we've taught a neural network to take a single input datapoint and predict a single output datapoint. However, we're trying to build a neural network that tells us "whether or not it is safe to cross the street". We need it to know more than one streetlight! How do we do this? Well... we train it on all the streetlights at once!

```
import numpy as np

weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [[ 1, 0, 1 ],
                          [ 0, 1, 1 ],
                          [ 0, 0, 1 ],
                          [ 1, 1, 1 ],
                          [ 0, 1, 1 ],
                          [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] # [1,0,1]
goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop"

for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
        goal_prediction = walk_vs_stop[row_index]

        prediction = input.dot(weights)

        error = (goal_prediction - prediction) ** 2
        error_for_all_lights += error

        delta = prediction - goal_prediction
        weights = weights - (alpha * (input * delta))
        print  "Prediction:" + str(prediction)
    print "Error:" + str(error_for_all_lights) + "\n"
```

It's actually pretty simple. We just created another for loop that iterates through each one of our streetlights... predicting, comparing to error, and updating our weights... And as we'll see... our error still goes to zero! (meaning it's correctly predicting all the lights!)

```
Error:2.6561231104
Error:0.962870177672
...
Error:0.000614343567483
Error:0.000533736773285
```

Write it from Memory!

Take time to write the "Polished Version" from memory.

Unless you've worked with a matrix library before, the conventions of numpy might feel a little foreign to you. Take the time to write the "Polished Version" from memory before continuing on to the next chapter. Doing so will solidify the concepts in your mind (ensuring you didn't miss one), help you feel comfortable with what we're building, and most importantly, make you familiar with the basic conventions of matrix operation (which I will expect you to have in the following chapters). I can't begin to describe how valuable this exercise was for me the first time I did it, and I still write this particular network from memory on a semi-quarterly basis. I think you'll find it well worth your time to do so.

Building Your First "Deep" Neural Network
Introduction to Backpropagation

5

IN THIS CHAPTER

Why you should learn deep learning

Why you should read this book

What you need to get started

We'll talk some about what kind of coding experience you need, and why I'm using Python for this book.

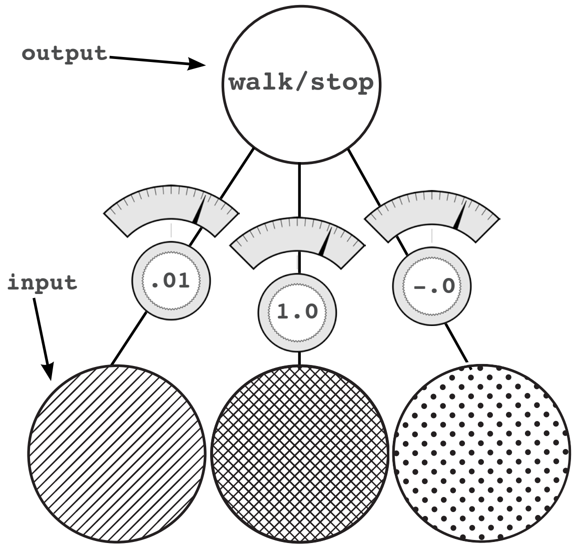
“ This is a sample quote that I'm trying to make, but Andrew... would you like to perhaps put something else here? ”

— *SOME AUTHOR*

Neural Networks Learn Correlation

What did our last neural network learn?

We just got done training a single-layer neural network to take a streetlight pattern and identify whether or not it was safe to cross the street. Let's take on the neural network's perspective for a moment. The neural network doesn't know that it was processing streetlight data. All it was trying to do was identify *which* input (out of the 3 possible) correlated with the output. You can see that it correctly identified the middle light by analyzing the final weight positions of the network.



Notice that the middle weight is very near 1 while the far left and right weights are very near 0. At a high level, all the iterative, complex processes for learning we identified actually accomplished something rather simple. The network *identified correlation* between the middle input and output. The correlation is located wherever the weights were set to high numbers. Inversely, *randomness* with respect to the output was found at the far left and far right weights (where the weight values are very near 0).

How did it identify correlation? Well, in the process of **Gradient Descent**, each training example either asserts *up pressure* or *down pressure* on our weights. On average, there was more *up pressure* for our middle weight and more *down pressure* for our other weights. Where does the pressure come from? Why is it different for different weights?

Up and Down Pressure

It comes from our data.

Each neuron is individually trying to correctly predict the output given the input. For the most part, it ignores all the other neurons when attempting to do so. The only *cross communication* occurs in that all three weights must share the same error measure. Our *weight update* is nothing more than taking this *shared* error measure and multiplying it by each *respective* input. Why do we do this? Well, a key part of why neural networks learn is by **error attribution**, which means that given a shared error, the network needs to figure out which weights contributed (so they can be adjusted) and which weights did NOT contribute (so they can be left alone).

Training Data

1	0	1		0
0	1	1	→	1
0	0	1		0
1	1	1	→	1
0	1	1	→	1
1	0	1		0

Weight Pressure

-	0	-		0
0	+	+	→	1
0	0	-		0
+	+	+	→	1
0	+	+	→	1
-	0	-		0

Consider the first training example. Because the middle input is a 0, then the middle weight is *completely irrelevant* for this prediction. No matter what the weight is, it's going to be multiplied by zero (the input). Thus, any error at that training example (regardless of whether its too high or too low), can only be **attributed** to the far left and right weights.

Consider the pressure of this first training example. If the network should predict 0, and two inputs are 1s, then this is going to cause error which drives the weight values *towards 0*.

The "Weight Pressure" table helps describe the affect that each example has on each respective weight. + indicates that it has pressure *towards 1* whereas the - indicates that it has pressure *towards 0*. Zeroes (0) indicate that there is *no pressure* because the input datapoint is 0, so that weight won't be updated. Notice that the far left weight has 2 negatives and 1 positive, so on average the weight will move towards 0. The middle weight has 3 positives, so on average the weight will move towards 1.

Up and Down Pressure (cont.)

Training Data

1	0	1	0
0	1	1	1
0	0	1	0
1	1	1	1
0	1	1	1
1	0	1	0

Weight Pressure

-	0	-	0
0	+	+	1
0	0	-	0
+	+	+	1
0	+	+	1
-	0	-	0

So, each individual weight is attempting to compensate for error. In our first training example, we see *discorrelation* between the far right and left inputs and our desired output. This causes those weights to experience *down pressure*. This same phenomenon occurs throughout all 6 training examples, rewarding correlation with pressure *towards 1* and penalizing de-correaltion with pressure *towards 0*. On average, this causes our network to find the correlation that is present between our middle weight and the output to be the dominant predictive force (heavest weight in the weighted average of our input) making our network quite accurate.

Bottom Line

Our prediction is a weighted average of our inputs. Our learning algorithm rewards inputs that correlate with our output with *upward pressure* (towards 1) on their weight while rewarding inputs with no-correlation with *downward pressure*. So that our weighted average of our inputs will find perfect correlation with our output.

Now, the mathematician in you might be cringing a little bit. "upward pressure" and "downward pressure" are hardly precise mathematical expressions, and they have plenty of edge cases where this logic doesn't hold (which we'll address in a second). However, we will later find that this is an *extremely* valuable approximation, allowing us to temporarily overlook all the complexity of Gradient Descent and just remember that *learning rewards correlation* with larger weights or more generally *learning finds correlation between our two datasets*.

Edge Case: Overfitting

Sometimes correlation happens accidentally...

Consider again the first example in the training data. What if our far left weight was 0.5 and our far right weight was -0.5. Their prediction would equal 0! The network would predict perfectly! However, it hasn't remotely learned how to safely predict streetlights. This phenomenon is known as **overfitting.**

Deep Learning's Greatest Weakness: Overfitting

Error is shared between all of our weights. If a particular configuration of weights *accidentally* creates perfect correlation between our prediction and the output dataset (such that error == 0) without actually giving the heaviest weight to the best inputs... the neural network will stop learning.

In fact, if it wasn't for our other training examples, this fatal flaw would cripple our neural network. What do the other training examples do? Well, let's take a look at the second training example. It would bump the far right weight *upward* while not changing the far left weight. This throws off the equilibrium that stopped the learning in our first example. So, as long as we don't train *exclusively on the first example*, the rest of the training examples will help the network avoid getting stuck in these edge case configurations that exist for any one training example.

This is super, super important. Neural networks are so flexible that they can find many, many different weight configurations that will correctly predict for a subset of your training data. In fact, if we trained our neural network on the first 2 training examples, it would likely stop learning at a point where it did NOT work well for our other training examples. In essence, it *memorized* the two training examples instead of actually finding the *correlation* that will *generalize* to any possible streetlight configuration.

If we only train on two streetlights... and the network just finds these edge case configurations... it coudl FAIL to tell us whether it is safe to cross the street when it sees a streetlight that wasn't in our training data!

The greatest challenge you will face with deep learning is convincing your neural network to *generalize* instead of just *memorize*. We will see this again.

Edge Case: Conflicting Pressure

Sometimes correlation fights itself.

Consider the far right column in the "Weight Pressure" table below. What do you see? This column seems to have an equal number of *upward* and *downward* pressure moments. However, we have seen that the network correctly pushes this (far right) weight down to 0 which means that the *downward* pressure moments must be larger than the *upward* ones. How does this work?

Training Data

1	0	1		0
0	1	1	→	1
0	0	1		0
1	1	1	→	1
0	1	1	→	1
1	0	1		0

Weight Pressure

–	0	–		0
0	+	+	→	1
0	0	–		0
+	+	+	→	1
0	+	+	→	1
–	0	–		0

The left and middle weights have enough signal to converge on their own. The left weight falls to 0 and the middle weight moves towards 1. As the middle weight moves higher and higher, the error for positive examples continues to decrease. However, as they approach their optimal positions, the de-correlation on the far right weight becomes more apparent. Let's consider the extreme example of this, where the left and middle weights are perfectly set to 0 and 1 respectively. What happens to our network? Well, if our right weight is above 0, then our network predicts too *high* and if our right weight is beneath 0, our network predicts too *low*.

In short, as other neurons learn, they absorb some of the *error*. They absorb some part of the *correlation*. They cause the network to predict with *moderate* correlative power which reduces the error. The other weights then only try to adjust their weights to correctly predict what's left! In this case, because the middle weight has consistent signal to absorb *all* of the correlation (because of the 1:1 relationship between the middle input and the output), the error when we want to predict 1 becomes very *small* but the error to predict 0 becomes large... pushing our middle weight downward.

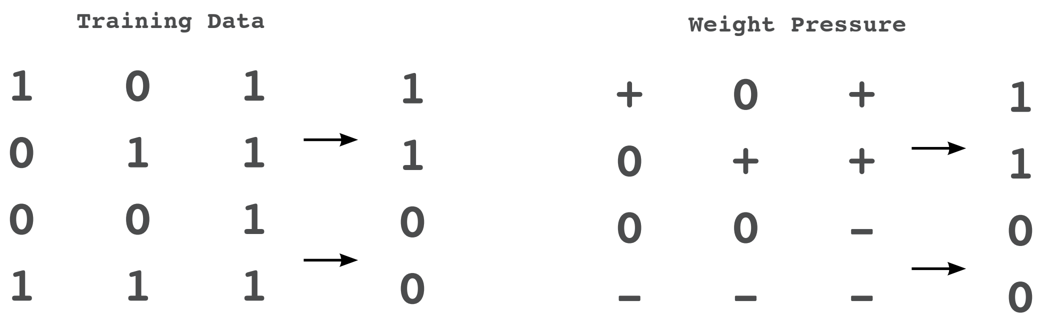
Edge Case: Conflicting Pressure (cont.)

It doesn't always work out like this.

In some ways, we kindof got lucky. If our middle node hadn't been so perfectly correlated, our network might have struggled to silence our far right weight. In fact, later we will learn about **Regularization** which forces weights with conflicting pressure to move towards 0.

This is advantageous because if a weight has equal pressure *upward* and *downward*, then it isn't really good for anything. It's not helping either direction. In essence, regularization aims to say "only weights with really strong correlation can stay on... everything else should be silenced because its contributing noise". It's sortof like natural selection... and as a side effect it would cause our neural network to train *faster* (fewer iterations) because our far right weight has this "both positive and negative" pressure problem. In this case, because our far right node isn't *definitively correlative*, the network would immediately start driving it towards 0. Without regularization (like we trained it before), we won't end up learning that the far right input is useless until after the left and middle start to figure their patterns out. More on this later.

So, if networks look for correlation between an input colum of data and our output column, what would our neural network do with this dataset?

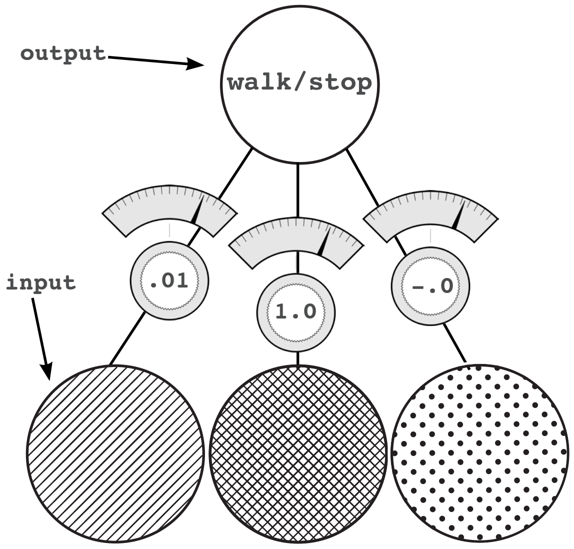


There is no correlation between any input column and the output column. Every weight has an equal amount of upward pressure as it does downward pressure. *This dataset is a real problem for our neural network.* Previously, we could solve for input datapoints that had *both* upward and downward pressure because other neurons would start solving for either the positive or negative predictions... drawing our balanced neuron to favor up or down. However, in this case, *all of the inputs* are equally balanced between *positive* and *negative* pressure. What do we do?

Learning Indirect Correlation

If your data doesn't have correlation... let's create intermediate data that does!

Previously, I have described neural networks as an instrument that searches for correlation between *input output datasets*. I should refine this just a touch. In reality, neural networks actually search for correlation between their input and output *layers*. We set the values of our input *layer* to be individual rows of our input data... and we try to train the network so that our output *layer* equals our output dataset. Funny enough... the neural network actually doesn't really "know" about data. It just searches for correlation between the input and output layers.



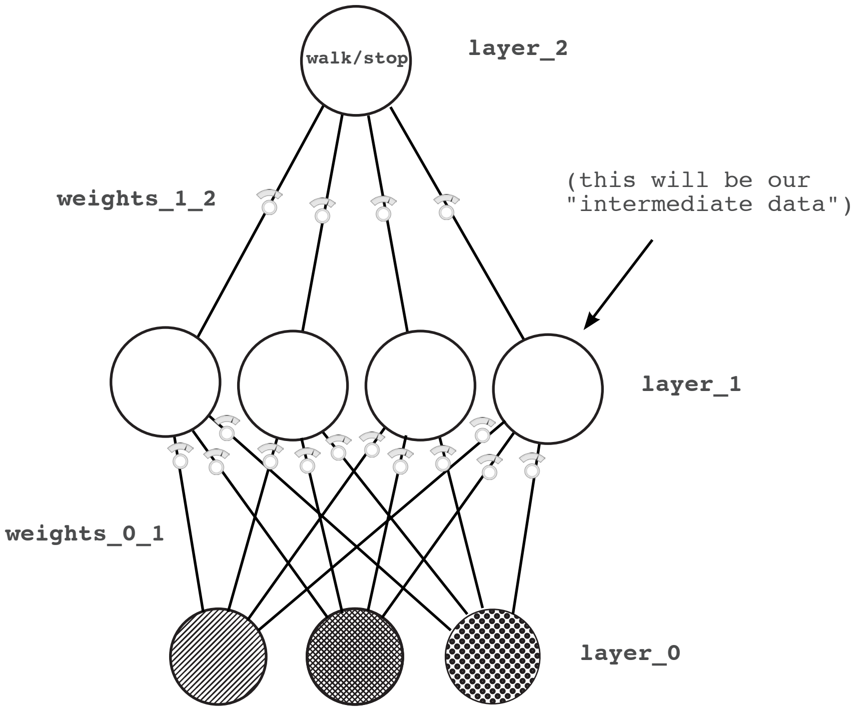
Unfortunately, we just encountered a new streetlights dataset where there isn't *any correlation* between our input and output. The solution is simple. Let's use *two* of these networks. The first one will create an *intermediate dataset* that has *limited* correlation with our output. The second will then use that *limited* correlation to correctly predict our output!

Since our input dataset doesn't correlate with our output dataset... we're going to use our input dataset to create an *intermediate dataset* that *DOES* have correlation with our output. It's kindof like cheating!

Creating Our Own Correlation

If your data doesn't have correlation... let's create intermediate data that does!

Below, you'll see a picture of our new neural network. Notice that we basically just stacked two neural networks on top of each other. The middle layer of nodes (layer_1) represents our *intermediate dataset*. Our goal is to train this network so that even though there's no correlation between our input dataset and output dataset (layer_0 and layer_2) that our layer_1 dataset that we create *using layer_0* will have correlation with layer_2.

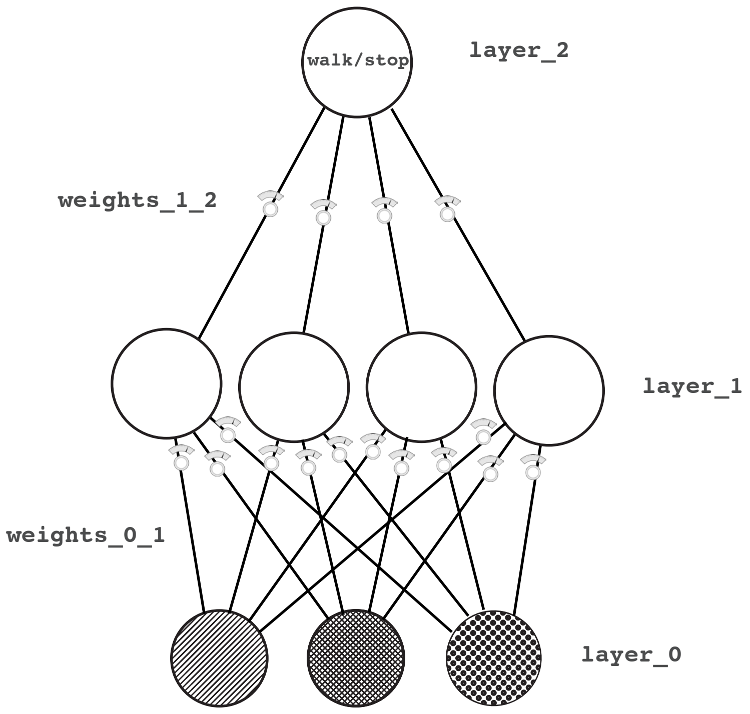


Things to notice: This network is still just a function! It still just has a bunch of weights that are collected together in a particular way. Furthermore, **Gradient Descent** still works because we can calculate how much each weight contributes to the error and adjust it to reduce the error to 0. That's exactly what we're going to do.

Stacking Neural Networks?

How does the prediction actually work?

So, when you look at the architecture below, the prediction occurs exactly as you might expect when I say "stack neural networks". The *output* of the first "lower" network (layer_0 to layer_1) is the *input* to the second "upper" neural network (layer_1 to layer_2). The prediction for each of these networks is identical to what we saw before.



So, as we start to think about how this neural network learns, we actually already know a great deal. If we ignored the lower weights and just considered their output to be our training set, then the top half of the neural network (layer_1 to layer_2) is just like the networks we trained in the last chapter. We can use all the same learning logic to help them learn. In fact, this is the case.

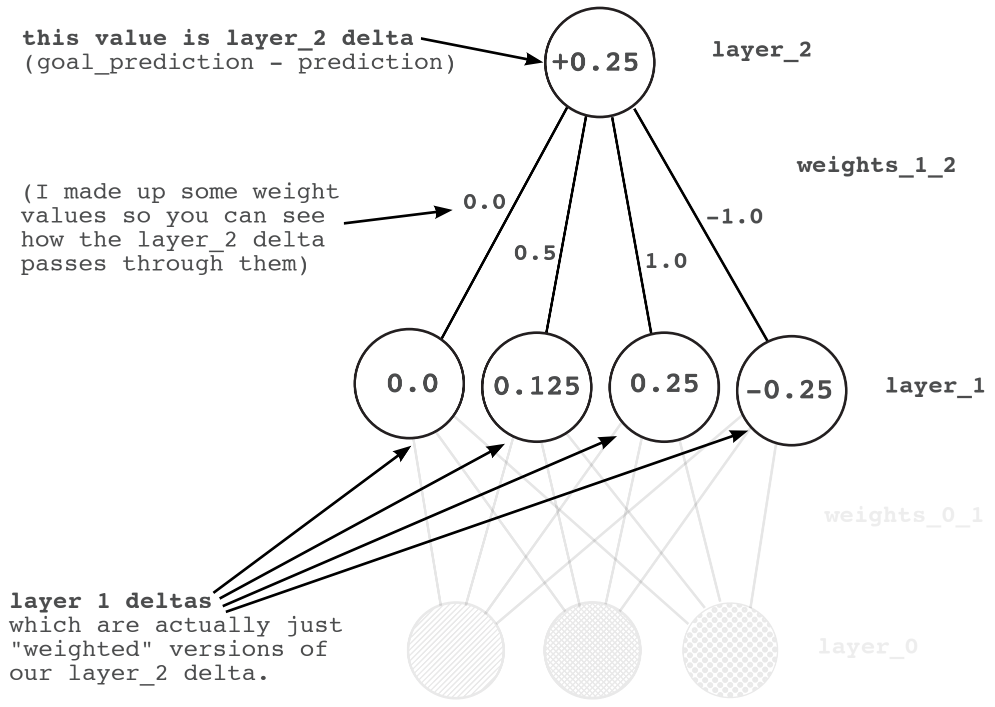
So, the part that we *don't yet understand* is how to update the weights between layer_0 and layer_1. What do they use as their error measure? If you remember from the last chapter, our *cached/normalized error measure* was called *delta*. In our case, we want to figure out how to know the *delta* values at layer_1 so that they can help layer_2 make accurate predictions.

Backpropagation: Long Distance Error Attribution

The "weighted average error"

What is the prediction from layer_1 to layer_2? It's just a weighted average of the values at layer_1. So, if layer_2 too high by "x" amount. How do we know which values at layer_1 contributed to the error? Well, the ones with *higher weights* (weights_1_2) contributed more! The ones with *lower weights* from layer_1 to layer_2 contributed less! Consider the extreme. Let's say that the far *left* weight from layer_1 to layer_2 was zero. How much did that node at layer_1 cause the network's error? ZERO!

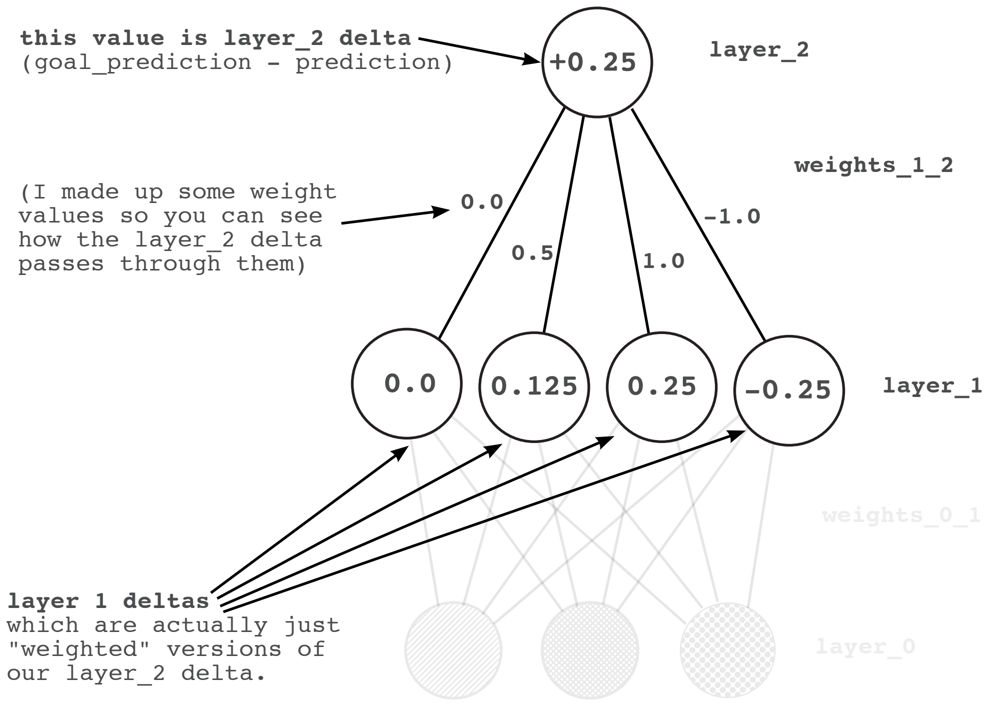
It's so simple it's almost hilarous. Our weights from layer_1 to layer_2 exactly describe how much each layer_1 neuron contributes to the layer_2 prediction. This means that those weights ALSO *exactly describe* how much each layer_1 neuron contributes to the layer_2 **error!** So, how do we use the delta at layer_2 to figure out the delta at layer_1? We just multiply it by each of the respective weights for layer_1!!! It's like our prediction logic in reverse! This process of "moving delta signal around" is called **backpropagation**.


this value is layer_2 delta
(goal_prediction - prediction)
+0.25
layer_2
weights_1_2
(I made up some weight values so you can see how the layer_2 delta passes through them)
0.0
0.5
1.0
-1.0
0.0
0.125
0.25
-0.25
layer_1
weights_0_1
layer 1 deltas which are actually just "weighted" versions of our layer_2 delta.
layer_0

Backpropagation: Why does this work?

The "weighted average delta"

In our neural network from the previous chapter, our *delta* variable told us "the **direction** and **amount** we want the value of this neuron to change next time". All backpropagation let us do is say "hey... if you want this neuron to be X amount higher... then each of these previous 4 neurons need to be X*weights_1_2 amount higher/lower... becuase these weights were *amplifying* the prediction by weights_1_2 times". When used in *reverse,* our weights_1_2 matrix *amplifies the error* by the appropriate amount. It *amplifies the error* so that we know how much each layer_1 node should move up or down. Once we know this, we can just update each weight matrix just like we did before. For each weight, multiply its output *delta* by its input *value*... and adjust our weight by that much. (or we can scale it with *alpha*).


this value is layer_2 delta
(goal_prediction - prediction)
+0.25
layer_2
weights_1_2
(I made up some weight values so you can see how the layer_2 delta passes through them)
0.0
0.5
1.0
-1.0
0.0
0.125
0.25
-0.25
layer_1
weights_0_1
layer 1 deltas which are actually just "weighted" versions of our layer_2 delta.
layer_0

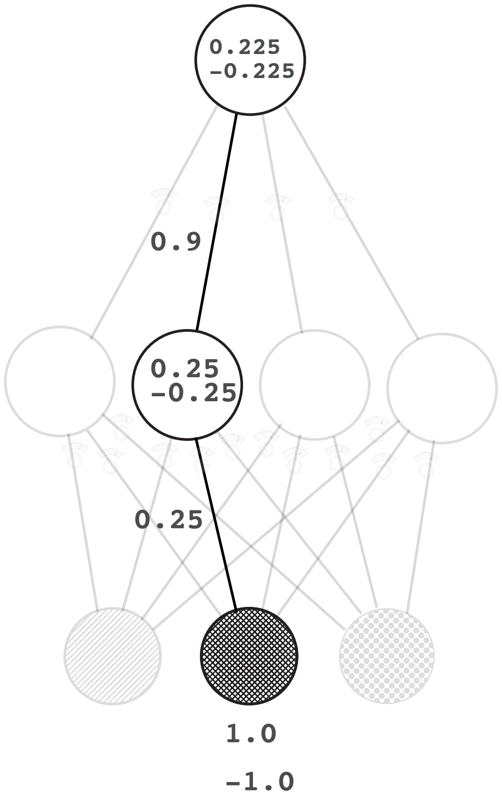
Linear vs Non-Linear

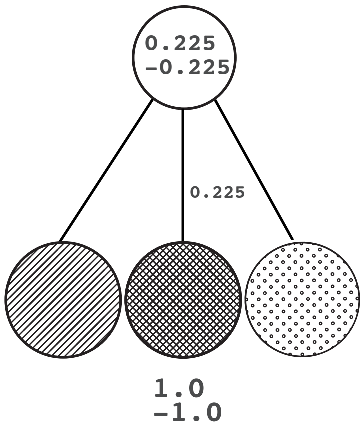
This is probably the hardest concept in the book. Let's take it slow.

I'm going to show you a phenomenon. As it turns out, we need one more "piece" to make this neural network train. We're going to take it from two perspectives. The first is gong to show you why *the neural network can't train without it*. In other words, first I'm going to show you why our neural network is currently broken. Then, once we add this piece, I'm going to show you what it does to fix this problem. For now, check out this simple algebra.

```
1 * 10 * 2 = 100          1 * 0.25 * 0.9 = 0.225
5 * 20 = 100              1 * 0.225 = 0.225
```

Here's the takeaway, for any *two multiplications* that I do, I can actually accomplish the *same thing* using a single multiplication. As it turns out, this is bad. Check out the following.





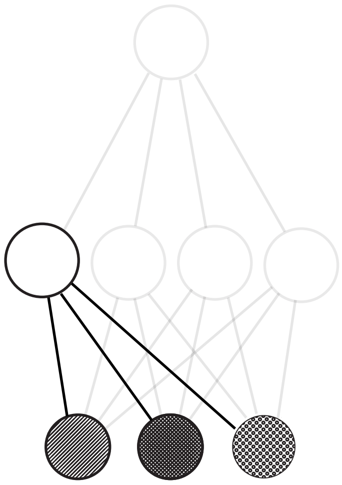
These two graphs show you *two training examples each*, one where the input is 1.0 and another where the input is -1.0. Here's the bottom line, **for any 3-layer network we create, there's a 2-layer network that has identical behavior**. It turns out that just stacking two neural nets (as we know them at the moment) doesn't actually give us any more power! Two consecutive weighted sums is just a more expensive version of one weighted sum.

Why The Neural Network Still Doesn't Work

If we trained the 3 layer network as it is now, it would NOT converge.

Problem: For *any* two consecutive *weighted sums* of the input, there exists a *single weighted sum* with exactly identical behavior. Aka... anything that our 3 layer network can do... our 2 layer network can also do.

Let's talk about the *middle layer* (layer_1) that we have at present (before we fix it). Right now, each node (out of the 4 we have), has a weight coming to it from each of the inputs. Let's think about this from a *correlation* standpoint. Each node in our *middle layer* subscribes to a *certain amount of correlation* with each input node. If the weight from an input to the middle layer is a 1.0, then it subscribes to *exactly* 100% of that node's movement. If that node goes up by 0.3, our middle node will follow. If the weight connecting our 0.5, it subscribes to *exactly* 50% of that node's movement.



The only way that our middle node can escape the correlation of one particular *input node* is if it *subscribes to additional correlation from another input node*. So, you can see, there's *nothing new* being contributed to this neural network. Each of our hidden nodes simply subscribe to a little bit of correlation from out input nodes. Our middle nodes don't actually get to *add anything to the conversation*. They don't get to have *correlation of their own*. They're just more or less correlated to various *input nodes*. However, since we KNOW that in our new dataset... there is NO correlation between ANY of our inputs and our output... then how could our middle layer help us at all?!?! It just gets to mix up a bunch of correlation that was already *useless!* What we really need is for our *middle layer* to be able to *selectively* correlate with our input. We want it to *sometimes* correlate with an input, and *sometimes not correlate*. That gives it correlation of it's own! This gives our middle layer the *opportunity* to not just "always be X% correlated to one input and Y% correlated to another input". Instead, it can be "X% correlated to one input.... only when it wants to be... but other times not be correlated at all!". This is "conditional correlation" or "sometimes correlation".

The Secret to "Sometimes Correlation"

We're going to simply turn our node "off" when the value would be below 0.

This might seem to simple to work, but consider this. If the node's value dropped below 0, normally the node would still have *just as much correlation* to the input as it always did! It would just happen to be negative. However, if we *turn off the node* (setting it to 0) when it would be negative, then it has *ZERO CORRELATION* to *ANY INPUTS* whenever it's negative.

What does this mean? It means that our node can now selectively pick and choose when it wants to be correlated to something. This allows it to say something like "make me perfectly correlated to the left input but ONLY when the right input is turned OFF". How would it do this? Well, if the weight from the left input is a 1.0, and the weight from the right input is a HUGE NEGATIVE NUMBER, then turning on both the left and right inputs would cause the node to just be 0 all the time. However, if just the left node was on, the node would take on the value of the left node.

This wasn't possible before! Before our middle node was either ALWAYS correlated to an input or ALWAYS not correlated! Now it can be conditional! Now it can speak for itself!

Solution: By turning any middle node off whenever it would be negative, we allow the network to *sometimes* subscribe to correlation from various inputs. This is *impossible* for 2-layer neural networks... thus adding power to 3-layer nets.

The fancy term for this "if the node would be negative then set it to 0" logic is called a **nonlinearity.** This is because without this tweak, our neural network is *linear*. Without this technique, our output layer only gets to pick from the same *correlation* that it had in the 2-layer network. It's still just subscribing to pieces of the *input layer*, which means that it can't solve our new streetlights dataset.

There are **many kinds of nonlinearities.** However, the one we discussed above is, in many cases, the best one to use. It's also the simplest.

For what it's worth, most other books/courses simply just say "consecutive matrix multiplication is still just a linear transformation". I find this to be very unintuitive. Furthermore, it makes it harder to understand what nonlinearities actually *contribute* and why you choose one over the other (which we'll get to later). It just says "without the nonlinearity two matrix multiplications might as well be 1". So, this page's explanation, while not the most common answer, is a better explanation of why we need nonlinearities (in my opinion).

A Quick Break

That last part probably felt a little abstract... that's totally ok. Let's chat for a sec.

So, here's the deal. In previous chapters we were working with very simple algebra. This meant that everything was ultimately grounded in fundamentally simple tools. This chapter has started building on the presmises we learned previously. In other words, previously we learned lessons like:

> We can compute the relationship between our *error* and any one of our *weights* so that we know how changing the weight changes the error. We can then use this to reduce our weight down to 0.

That was a **massive lesson!** However, now we're moving past it. Since we already worked through why that works, we can just trust it. We take the statement at face value. The next big lesson came at the beginning of this chapter:

> Adjusting our weights to reduce our error over a *series of training examples* ultimately just searches for correlation between our *input* and our *output* layers. If no correlation exists, then error will never reach 0.

This lesson is an even **bigger lesson!** Why? Well, it largely means that we can put the previous lesson out of our minds for now. We don't actually need it. Now we're focused on *correlation*. The takeaway for you is that you can't constantly think about *everything all at once*. You take each one of these lessons and you let yourself trust it. When it's a more *concise* summarization... a higher abstraction... of more granular lessons. We forget the granular and stick to the higher summarizations.

This is akin to a professional swimmer, biker, or really any other skill that requires a *combined fluid knowledge* of a bunch of really small lessons. A baseball player who *swings a bat* actually learned 1000s of little lessons to ultimately culminate in a great bat swing. However, he doesn't think of *all of them* when he goes to the plate! He just lets it be fluid... subcontious even. This is the same way for studying these math concepts.

Neural networks look for correlation between input and output... and you no longer have to worry about *how that happens*. We just know that it does. Now we're building on that idea! Let yourself relax and trust the things you've already learned.

Our New Prediction Code

What does our prediction logic look like?

What We Will Build

A Neural Network to Create Correlation

As per the other chapters, I want to start out by showing you what we will build, so that you can get a general sense for "where we're going". I've put in **bold** locations where we're doing something new that we'll be learning about. High level... it's sortof like stacking two neural networks from the previous chapter together, where the output of one is the input to the next. How do we train them together? Well, that's what we're about to learn.

```
import numpy as np

np.random.seed(1)

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

def sig2deriv(output):
    return output*(1-output)

alpha = 10
hidden_size = 4

streetlights = np.array( [[ 1, 0, 1 ],
                          [ 0, 1, 1 ],
                          [ 0, 0, 1 ],
                          [ 1, 1, 1 ],
                          [ 0, 1, 1 ],
                          [ 1, 0, 1 ] ] )

walk_vs_stop = np.array([[ 1, 1, 0, 0, 1, 1 ]]).T

weights_0_1 = 2*np.random.random((3,hidden_size)) - 1
weights_1_2 = 2*np.random.random((hidden_size,1)) - 1

for j in xrange(6000):

    layer_0 = streetlights

    layer_1 = sigmoid(np.dot(layer_0,weights_0_1))
    layer_2 = sigmoid(np.dot(layer_1,weights_1_2))

    layer_2_error = np.sum((layer_2 - walk_vs_stop) ** 2)

    layer_2_delta = (walk_vs_stop - layer_2) * sig2deriv(layer_2)
    layer_1_delta = l2_delta.dot(weights_1_2.T) * sig2deriv(layer_1)

    weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
    weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

    if(j % 1000 == 999):
        print "Error:" + str(layer_2_error)
```

this function forces x to be between 0 and 1 aka... a percentage

just like everything in neural networks, in order to learn we need to know the "derivative" so that we can adjust weights appropriately. this is the derivative of our sigmoid function. It cleverly uses the output of the sigmoid to compute its derivative... very clever

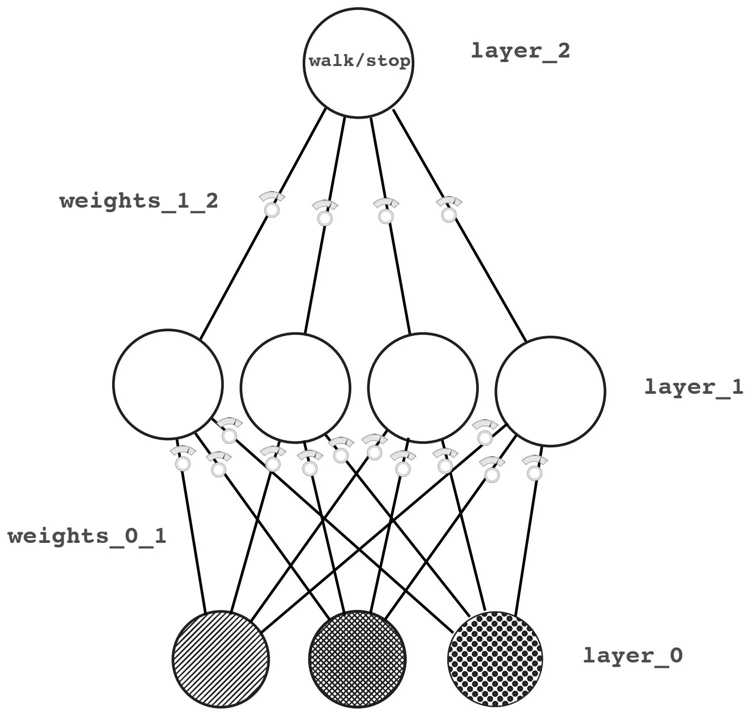
2 sets of weights now to connect our 3 layers (randomly initialized)

the output of the first prediction is the input to the second (layer_2)

we have to move error deltas from the output "through the network" to all 3 layers

What It Looks Like

Here's a picture of our neural network...

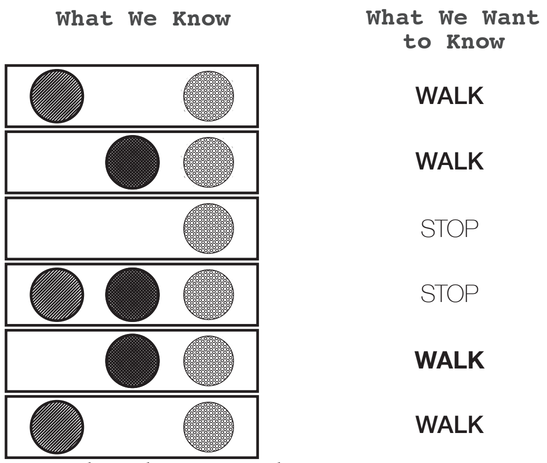


A Non-Linear Problem

A bit more difficult than our first streetlight problem.

In the last chapter, we used a neural network to learn a **direct relationship** between the middle streetlight and the "walk" pattern. This took the form of a direct link (weight) from our middle node to our "walk" node. In this chapter, we're going to teach our network to map an **indirect relationship** wherein it must *combine multiple inputs* to predict an output.

Let's take a look at the new pattern we want to learn to the right of this paragraph. Notice how in this case the "WALK" is only true when *either* the left or middle streetlights are turned on, but *not* when they are both turned on or off. In this way, *no single streetlight indicates whether it is safe to walk*. Only by studying a combination of the first and second streetlights can the neural network make an accurate prediction. This also means that there is ***no correlation*** between any of the inputs and the desired outputs. In fact, our previous network couldn't learn it.



Let's take a moment on vocabulary. In machine learning, when we are trying to map a 1:1 relationship (e.g. a direct relationship, direct correlation between two variables), we call this a **linear** problem. It represents a *linear relationship* between our input data and our output data. It can be solved by *linear* algorithms such as the neural network in the previous chapter. A **non-linear** problem, in turn, must *combine multiple inputs* to predict an output. Only certain *configurations of inputs* yield the pattern we are searching for.

To illustrate the difference between linear and nonlinear relationships, consider this cat picture. Specifically, consider the pixels in the cat's left ear. If we wanted to classify whether those pixels were turned on, a linear classifie could accurately classify this because it's simply the direct correlation between those particular pixels and the classification of "yes" or "no". However, the classification of whether or not there is a *cat in the picture* is non-linear because only certain configurations of pixels create the form of a cat. No individual pixel being on or off indicates the presence of a cat. It is the need to correctly model **arbitrary configurations** of inputs that give rise to the need for non-linearities.

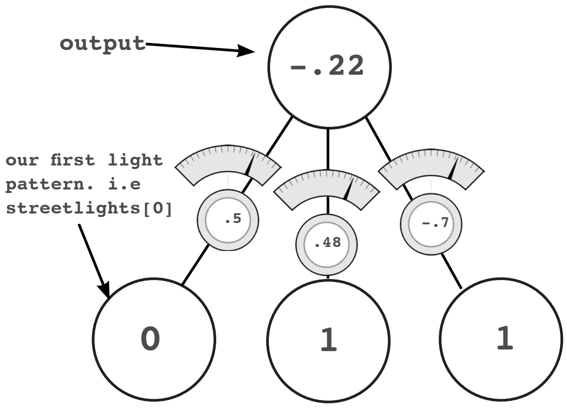
A Non-Linear Solution

A bit more powerful algorithm than our first neural network.

So, given the immense domain of problems that are **nonlinear**, how can we modify the neural network from the previous chapter to be able to search for the relationship between our new streetlight dataset and our new stop/walk pattern? We are going to be leveraging two new tools to do this. The first is an extension of our first neural network called a **layer**, and the second is a curvy line called a **sigmoid function**. Let's get started.

Improvement 1: Adding a Hidden Layer

Before we add our layer, let's first discuss a bit more about why we need to add this layer by looking at why our previous neural network can't solve this neural networks' problem. Our first neural net looked like this, and when we trained it, it ended up modifying the middle dial to equal **one** and all of the other dials to equal **zero**. Why? It did so because the original dataset had a direct correlation between the middle streetlight and safe walking. However, our new dataset that we're trying to learn has **no such correlation**. Stop and take a second look! Again, the new, nonlinear dataset has a pattern with significantly more complexity.


output
-.22
our first light pattern. i.e streetlights[0]
.5
.48
-.7
0
1
1