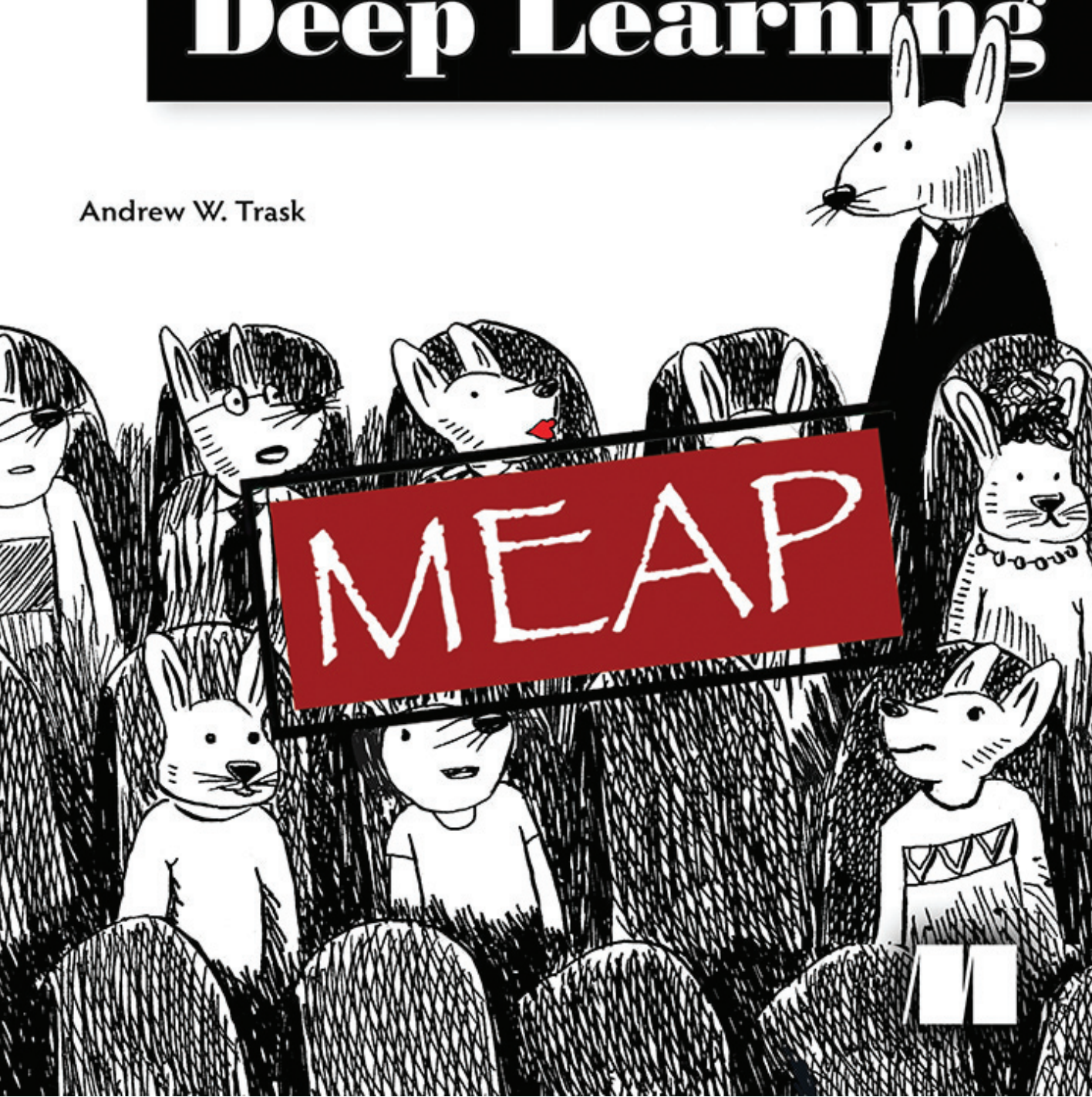# grokking

Deep Learning

Andrew W. Trask

MEAP

MEAP Edition
Manning Early Access Program
Grokking Deep Learning
Version 10

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to

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 prerequisite 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 like Torch, TensorFlow, or Keras.

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 few 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 disruption to 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 fortunately, 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 enormous 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. Even though Deep Learning is still an actively developing field with many challenges, recent developments have 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.

Will this be hard to learn?

How hard will you have to work before there is a "fun" payoff?

This is my favorite question. My definition of a "fun" payoff is the experience of witnessing something that I built *learning.* There's just something amazing about seeing a creation of your hands do something like that. If you also feel this way, then the answer is simple. A few pages into chapter 3, you'll create your first neural network. The only work involved between now and then is reading the pages between here and there.

After chapter 3, you may be interested to know that the *next* fun payoff occurs after you have memorized a small snippet of code and proceeded to read to the midway of Chapter 4. Each chapter will continue to work this way. Memorize a small code segment from the previous chapter, read the next chapter and then experience the payoff of a new learning neural network.

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.

Why you should read this book (cont.)

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 be prepared to 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 to 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. I will be building these examples in Python 2.7, but will test them for Python 3 as well. For easy installation, I also recommend the Anaconda framework (https://docs.continuum.io/anaconda/install).

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 only 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 Deep Learning 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! For Deep Learning, 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 (Python 2.7 is what I'm working in). 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?

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

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 Chairman, 2016)

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.
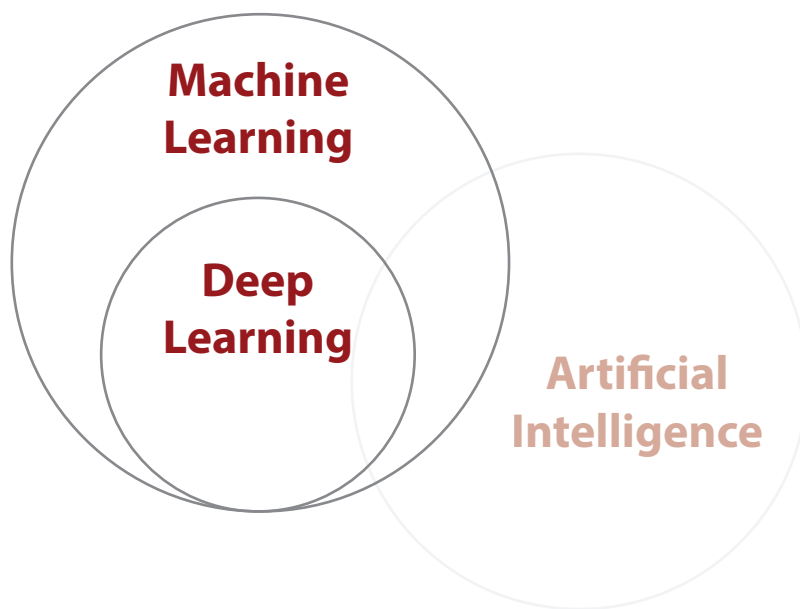
Notice in the figure above that not all of Deep Learning is focused around pursuing Generalized Artificial Intelligence (i.e. sentient machines like in the movies). In fact, many applications of this technology are applied to solve a wide variety of problems in industry. This book seeks to focus on learning the fundamentals of Deep Learning behind both cutting edge research and industry, helping to prepare you for either.

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 imitate it in some way which can be either direct or indirect **imitation**.

<div align="center">

machine learning ~= monkey see monkey do

</div>

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. This would be profound.

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.

Monday stock prices → supervised learning → Tuesday stock prices

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 (i.e. "Narrow AI"). 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.

what we know → supervised learning → what we want to know

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 it is in. Thus, your dataset turns from a bunch of datapoints into a bunch of labels. Why are the labels 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*. We will discover more on this later in the book.


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 form 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:
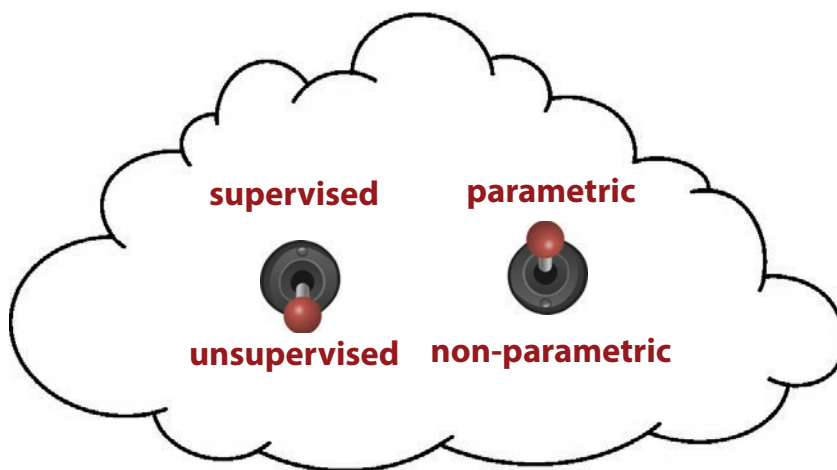
As you can see, we really have four different types of algorithm to choose from. An algorithm is either unsupervised or supervised and it is either parametric or non-parametric. Whereas the previous section on supervision is really about the **type of pattern** being learned, parametricism is about the 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).

As an example, 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 (parametric). A teenager, however, might just count the number of sides (4) and then search for the hole with an equal number (non-parametric). Parametric models tend to use "trial and error", where non-parametric models tend to "count". Let's look closer.

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*.
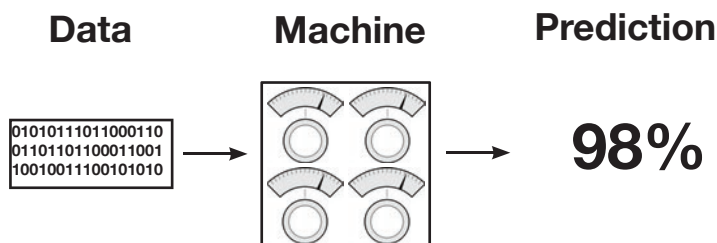
Learning is accomplished by turning the knobs to different angles. If we're trying to predict the probability that the Red Socks will win the World Series, then this model would first take data (such as sports stats like win/loss record or average number of toes) and make a prediction (such as 98% chance). 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.**
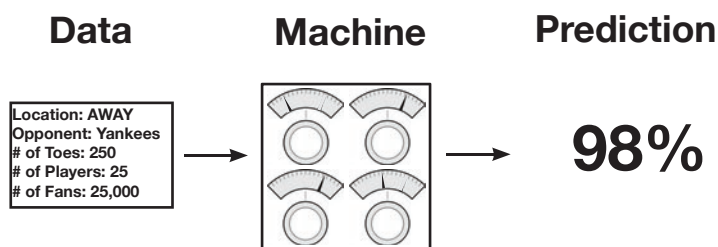
Perhaps it would "turn up" the "win/loss record" knob if the team's win/loss record was a good predictor. Inversely, it might turn down the "average number of toes" knob if that datapoint wasn't a good predictor. This is how parametric models learn!

Note that the entirety of what the model has learned can be captured in the positions of the knobs at any given time. One can also think of this type of learning model as a search algorithm. We are "searching" for the appropriate knob 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 into its three steps.

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.

Data   **Machine**   **Prediction**

Location: AWAY
Opponent: Yankees
# of Toes: 250   →   →   **98%**
# of Players: 25
# of Fans: 25,000

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 knobs by studying both how **much** the model missed (98%) and what the input data <u>was</u> (sports stats) at the time of prediction. It then turns the knobs 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 <u>*prediction's sensitivity to different types of input data*</u>. That's what we're changing when we "learn".

```
Adjusting Sensitivity
  By Turning Knobs
```
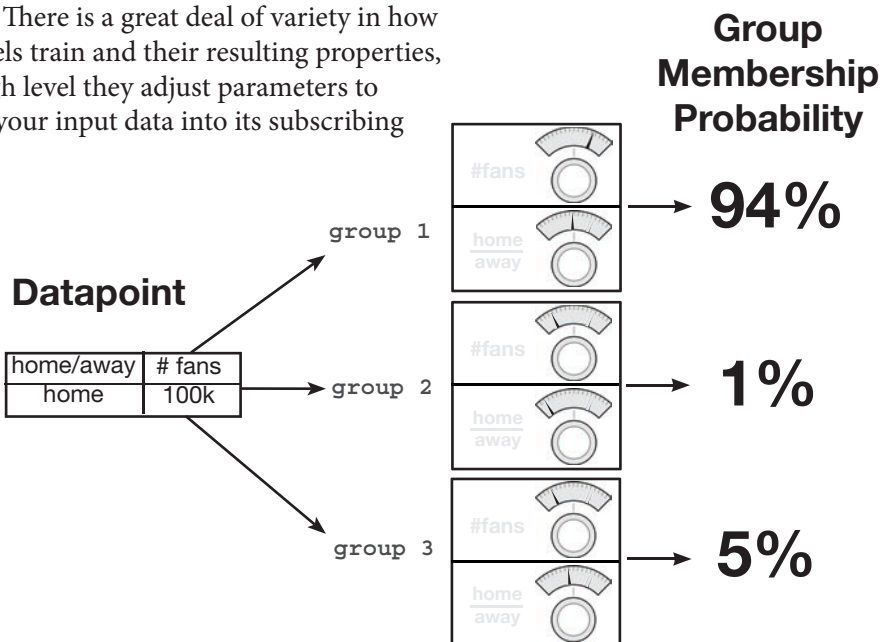
win/loss   home/away

#toes   #fans
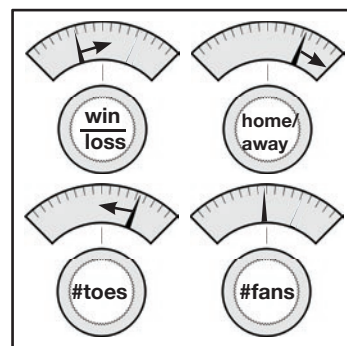
Unsupervised Parametric Learning

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 maps 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).

Group Membership Probability

Datapoint

home/away	# fans
home	100k

group 1 → 94%

group 2 → 1%

group 3 → 5%

Non-Parametric Learning

Oversimplified: Counting based methods

Non-Parametric learning is a class of algorithm wherein the number of parameters is based on data (instead of pre-defined). This lends itself to methods that generally count in one way or another, thus increasing the number of parameters based on the number of items being counted within the data. In the supervised setting, for example, a non-parametric model might count the number of times a particular color of streetlight causes cars to "go". After counting only a few examples, this model would then be able to predict that *middle* lights always (100%) cause cars to "go" and *right* lights only sometimes (50%) cause cars to "go".

Notice that this model would have 3 parameters, 3 counts indicating the number of times each colored light turned on and cars "go" (perhaps divided by the number of total observations). If there had been 5 lights, there would have been 5 counts (5 parameters). What makes this simple model *non-parametric* is this trait wherein the number of parameters changes based on the data (in this case, the number of lights). This is in contrast to parametric models, which start with a set number of parameters and, more importantly, can have more or less parameters purely at the disgretion of the scientist training the model (irrespective of data).

A close eye might question this idea. Our parametric model from before seemed to have a knob for each input datapoint. In fact, most parametric models still have to have some sort of *input* that is based on the number of classes in the data. Thus you can see that there is a bit of *grayness* between parametric and non-parametric algorithms. Even parametric algorithms still are somewhat influenced by the number of classes in the data, even if they are not explicitly counting patterns.

Perhaps this also illumates that *parameters* is actually a very generic term, referring only to the set of numbers used to model a pattern (without any limitation on how those numbers are used). Counts are parameters. Weights are parameters. Normalized variants of counts or weights are parameters. Correlation coefficients can be parameters. It's simply referring to the set of numbers used to model a pattern. As it happens, Deep Learning is a class of parametric models. We won't be discussing non-parametric models further in this book, but they are a very interesting and powerful class of algorithm.

Conclusion

In this chapter, we have gone a level deeper into the various flavors of Machine Learning. We have learned that a Machine Learning algorithm is either Supervised or Unsupervised and either Parametric or Non-Parametric. Furthermore, we have explored exactly what makes these 4 different groups of algorithms distinct. We have learned that Supervised Machine Learning is a class of algorithm where we learn to predict one dataset given another and that Unsupervised Learning generally groups a single dataset into various kinds of clusters. We learned that Parametric algorithms have a fixed number of *parameters* and that Non-Parametric algorithms adjust the number of parameters they have based on the dataset.

Deep Learning leverages Neural Networks to perform both supervised and unsupervised prediction. Up until now, we have stayed at mostly a conceptual level, gaining our bearings on the field as a whole and our place in it. In the next chapter, we will be building our first neural network, and all subsequent chapters will be *project based*. So, pull out your Jupyter notebook and let's jump right in!

Introduction to Neural Prediction
Forward Propagation

<div style="text-align:right">**3**</div>

IN THIS CHAPTER ·······························

> "
> I try not to get involved in the business of
> prediction. It's a quick way to look like an idiot.
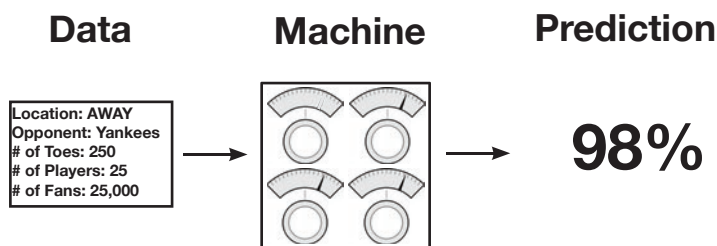> "
>
> — *WARREN ELLIS*

Step 1: Predict

This chapter is about "Prediction"

In the previous chapter, we learned about the paradigm: "Predict, Compare, Learn". In this chapter, we will dive deep into the first step, "Predict". You may remember that the predict step looks a lot like this.


In this chapter, we're going to learn more about what these 3 different parts of a neural network prediction really look like under the hood. Let's start with the first one, the Data. In our first neural network, we're going to predict one datapoint at a time, like so.
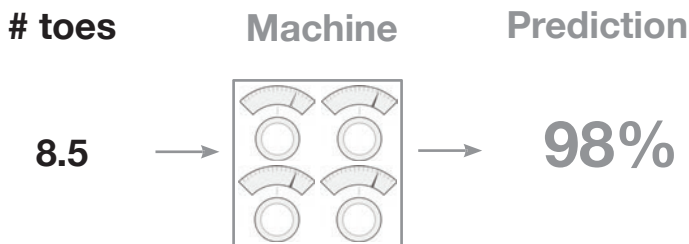

Later on, we will find that the "number of datapoints at a time" that we want to process will have a significant impact on what our network looks like. You might be wondering, "how do I choose how many datapoints to propagate at a time?" The answer to this question is based on whether or not you think the neural network can be accurate with the data you give it. For example, if I'm trying to predict whether or not there's a cat in a photo, I definitely need to show my network all the pixels of an image at once. Why? Well, if I only sent you one pixel of an image, could you classify whether the image contained a cat? Me neither! (That's a general rule of thumb by the way. Always present enough information to the network, where "enough information" is defined loosely as how much a human might need to make the same prediction).

Let's skip over the network for now. As it turns out, we can only create our network once we understand the shape of our input and output datasets (For now, shape means "number of columns" or "number of datapoints we're processing at once"). For now, we're going to stick with the "single-prediction" of "likelihood that the baseball team will win".

 # toes   **Machine**   **Win Probability**

8.5 $\longrightarrow$ $\longrightarrow$ **98%**
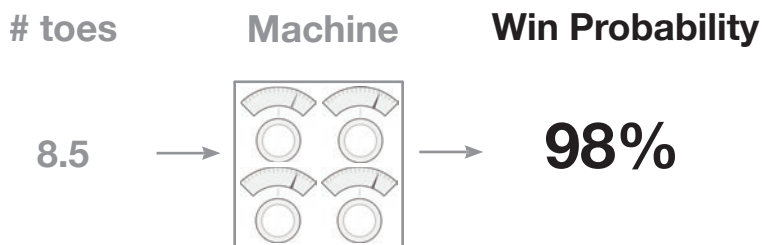
Ok, so now that we know that we want to take one input datapoint and output one prediction, we can create our neural network. Since we only have one input datapoint and one output datapoint, we're going to build a network with a single knob mapping from the input point to the output. Abstractly these "knob"s are actually called "weight"s, and we will refer to them as such from here on out. So, without further ado, here's our first neural network with a single weight mapping from our input "#toes" to output "win?"

① **An Empty Network**

input data
enters here

predictions
come out here

**#toes** $\longrightarrow$ **win?**

.1
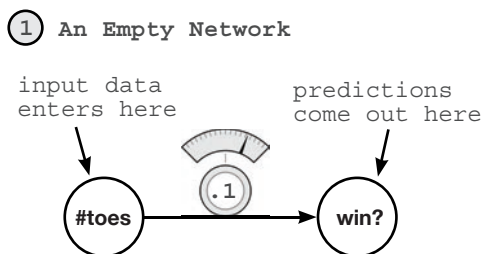
As you can see, with one weight, this network takes in one datapoint at a time (average number of toes on the baseball team) and outputs a single prediction (whether or not it thinks the team will win).

A Simple Neural Network Making a Prediction

Let's start with the simplest neural network possible.
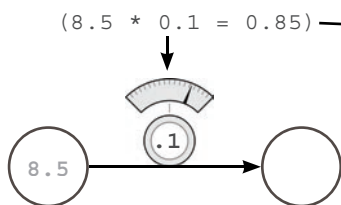
---

(1) An Empty Network

input data enters here

predictions come out here

.1

#toes → win?
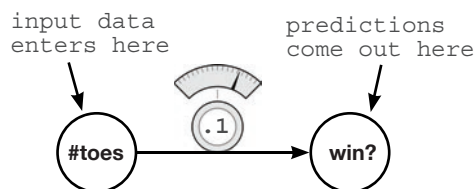
```
weight = 0.1
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

---

(2) Inserting One Input Datapoint
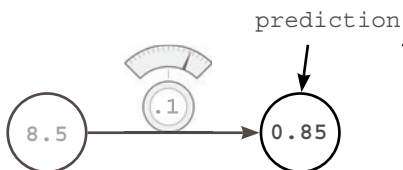
input data (#toes)

.1

8.5 →

```
number_of_toes = [8.5, 9.5, 10, 9]
input = number_of_toes[0]
pred = neural_network(input,weight)
print(pred)
```

---

(3) Multiplying Input By Weight

(8.5 * 0.1 = 0.85)

.1

8.5 →

```
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

---

(4) Depositing Prediction
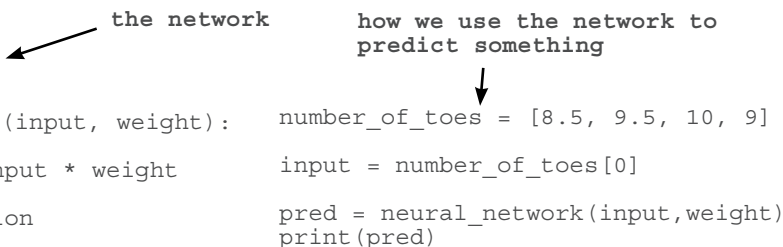
prediction

.1

8.5 → 0.85

```
number_of_toes = [8.5, 9.5, 10, 9]
input = number_of_toes[0]
pred = neural_network(input,weight)
```

What is a Neural Network?

This is a neural network.

Open up a Jupyter Notebook and run the following:

the network   **how we use the network to predict something**

```
weight = 0.1

def neural_network(input, weight):       number_of_toes = [8.5, 9.5, 10, 9]

    prediction = input * weight          input = number_of_toes[0]

    return prediction                    pred = neural_network(input,weight)
                                         print(pred)
```

You just made your first neural network and used it to predict! Congratulations! The last line prints the prediction (`pred`). It should be 0.85. So what is a neural network? For now, 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 `weight` to predict lower next time and vice 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 `weight` (up or down) to predict more accurately the next time it sees the same `input`.

What does this Neural Network do?

It multiplies the `input` **by a** `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, 8.5) and *multiplies* it by our `weight`. If the `weight` is 2, then it would *double our input*. If the `weight` is 0.01, then it would *divide* the input by 100. As you can see, some `weight` values make the input *bigger* and other values make it *smaller*.

---

① **An Empty Network**

input data
enters here

predictions
come out here

#toes → .1 → win?
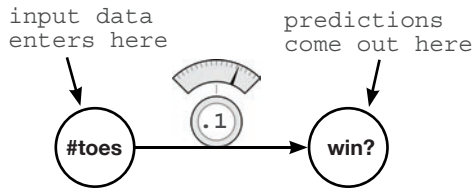
```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

The interface for our neural network is really quite simple. It accepts an `input` variable as *information*, and a weight variable as *knowledge* and outputs a `prediction`. Every neural network you will ever see works this way. It uses the *knowledge* in the weights to interpret the *information* in the input data. Later neural networks will accept larger, more complicated `input` and `weight` values, but this same underlying premise will always ring true.
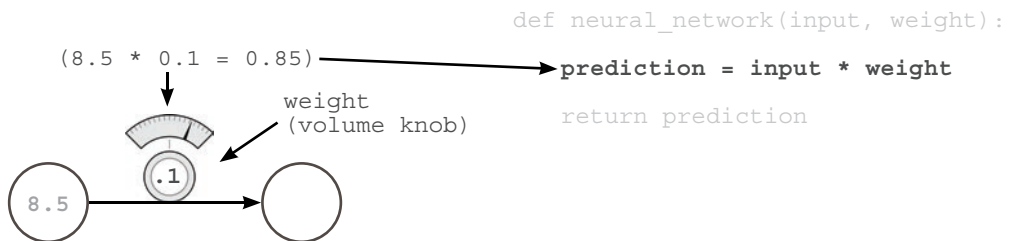
---

② **Inserting One Input Datapoint**

input data
(#toes)

8.5 → .1 →
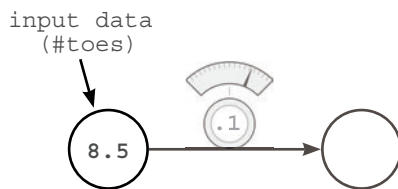
```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```
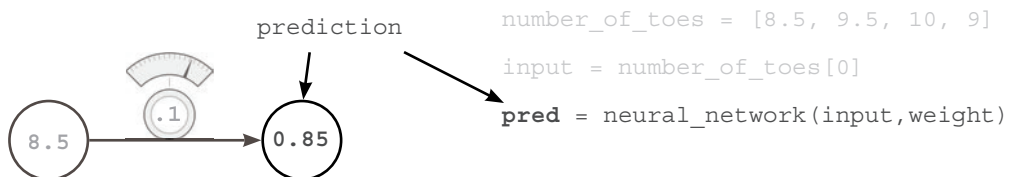

In this case, the "information" is the average number of toes on a baseball team before a game. Notice several things. The neural network does NOT have access to any information except *one* instance. If, after this prediction, we were to feed in number_of_toes[1], it would not remember the prediction it made in the last timestep. A neural network only knows what you feed it as input. It forgets everything else. Later, we will learn how to give neural networks "short term memories" by feeding in multiple inputs at once.

③ **Multiplying Input By Weight**

```
def neural_network(input, weight):
```

(8.5 * 0.1 = 0.85)

prediction = input * weight

```
    return prediction
```

weight
(volume knob)

.1

8.5

Another way to think about a neural network's `weight` is as a measure of *sensitivity* between the input of the network and its prediction. If the weight is very high, then even the tiniest input can create a really large prediction! If the 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. `weight` is a volume knob!

---

④ **Depositing Prediction**

prediction

```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]
```

pred = neural_network(input,weight)

.1

8.5

0.85

So in this case, what our neural network is really doing is applying a *volume knob* to our `number_of_toes` variable. In theory, this *volume knob* is able to tell us the likelihood that the team will win based on the average number of toes per player on our team. And this may or may not work. Truthfully, if the team had 0 toes, they would probably play terribly. However, baseball is much more complex than this. On the next page, we will present multiple pieces of information at the same time, so that the neural network can make more informed decisions.

Before we go, 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 "probability that people will wear coats today", if the temperature was -10 degrees Celsius, then a negative weight would predict a high probability that people would wear coats today.
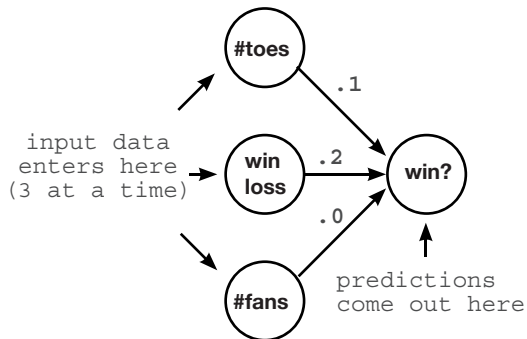
Temperature   Probability

-10   **89**

-8.9
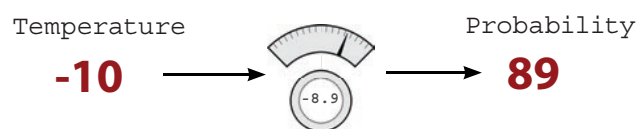
Making a Prediction with Multiple Inputs

Neural Networks can combine intelligence from multiple datapoints.

Our last neural network was able to take one datapoint as input and make one prediction based on that datapoint. Perhaps you've been wondering, "is average # of toes really a very good predictor?... all by itself?" If so, you're onto something. What if we were able to give our network more information (at one time) than just the "average number of toes". It should, in theory, be able to make more accurate predictions, yes? Well, as it turns out, our network can accept multiple input datapoints at a time. See the prediction below!
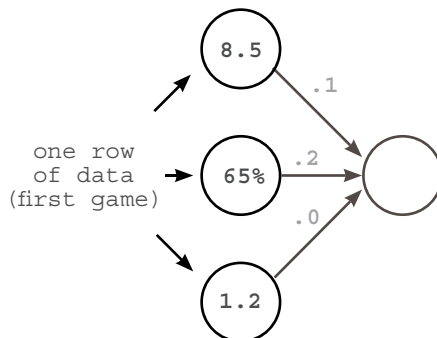
---

① An Empty Network With Multiple Inputs


```
weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

---

② Inserting One Input Datapoint


```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```
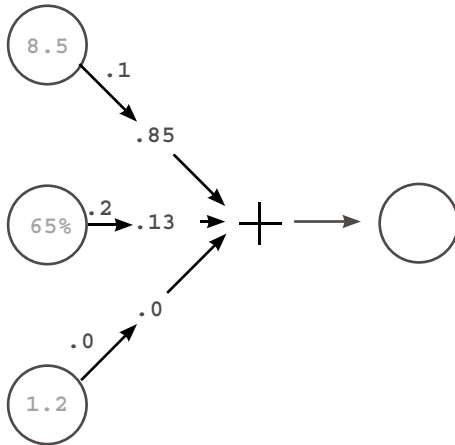
---

3 `Perform a Weighted Sum of Inputs`


```
def neural_network(input, weights):
    pred = w_sum(input,weights)
    return pred

def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(a):
        output += (a[i] * b[i])
    return output
```
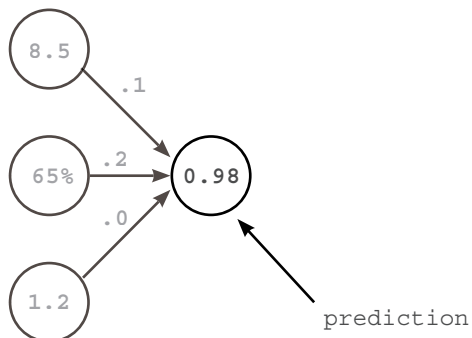
```
                      local
   inputs   weights   predictions
 ( 8.50  *  0.1 )  =  0.85  = toes prediction
 ( 0.65  *  0.2 )  =  0.13  = wlrec prediction
 ( 1.20  *  0.0 )  =  0.00  = fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction
      0.85       +       0.13      +       0.00       =     0.98
```

4 `Deposit Prediction`


```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```
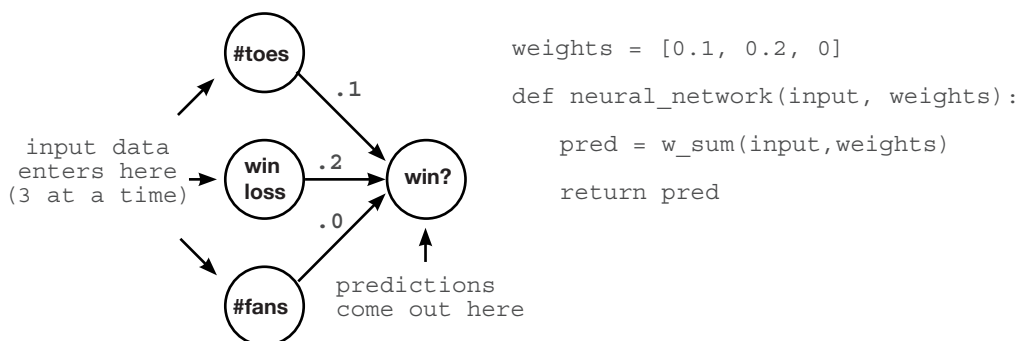
prediction

Multiple Inputs - What does this Neural Network do?

It multiplies 3 `inputs` **by 3** `knob_weights` **and sums them. This is a "weighted sum".**

At the end of the previous section, we came to realize the limiting factor of our simple neural network, it is only a volume knob on one datapoint. In our example, that datapoint was the average number of toes on a baseball team. We realized that in order to make accurate predictions, we need to build neural networks that can *combine multiple inputs at the same time*. Fortunately, neural networks are perfectly capable of doing so.

---

① `An Empty Network With Multiple Inputs`


```
weights = [0.1, 0.2, 0]
def neural_network(input, weights):
    pred = w_sum(input,weights)
    return pred
```


In this new neural network, we can accept *multiple inputs at a time* per prediction. This allows our network to combine various forms of information to make more well informed decisions. However, the fundamental mechanism for using our weights has not changed. We still take each input and run it through its own volume knob. In other words, we take each input and multiply it by its own weight. The new property here is that, since we have mutliple inputs, we have to sum their respective predictions. Thus, we take each input, multiply it by its respective weight, and then sum all the local predictions together. This is called a "weighted sum of the input" or a "weighted sum" for short. Some also refer to this "weighted sum" as a "dot product" as we'll see.

A Relevant Reminder

The interface for our neural network is quite simple. It accepts an input variable as *information*, and a weight variable as *knowledge* and outputs a prediction.
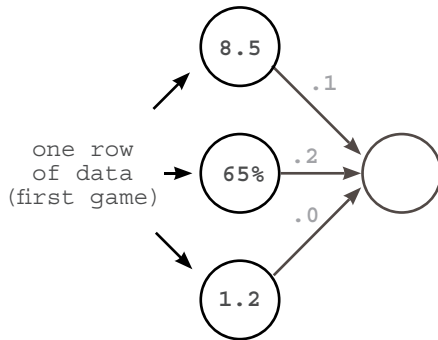
```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

This new need to process multiple inputs at a time justifies the use of a new tool. This tool is called a **vector** and if you've been following along in your iPython notebook, you've already been using it. A vector is nothing other than a *list of numbers.* `input` is a vector and `weights` is a vector. Can you spot any more vectors in the code above (there are 3 more)?

As it turns out, vectors are incredibly useful whenever you want to perform operations involving groups of numbers. In this case, we're performing a weighted sum between two vectors (dot product). We're taking two vectors of equal length (input and weights), multiplying each number based on its position (the first position in input is multiplied by the first position in weights, etc.), and then summing the resulting output.

It turns out that whenever we perform a mathematical operation between two vectors of equal length where we "pair up" values according to their position in the vector (again... position 0 with 0, 1, with 1, and so on), we call this an **elementwise** operation. Thus "elementwise addition" sums two vectors. "elementwise multiplication" multiplies two vectors.

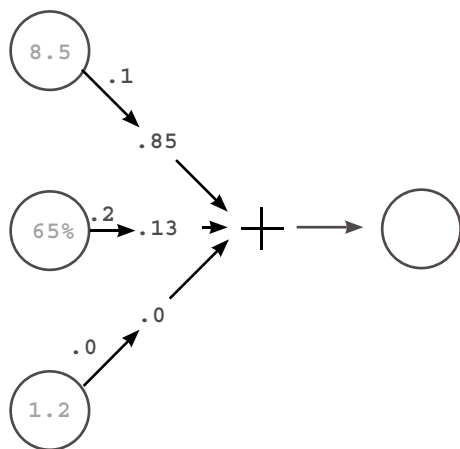### Challenge: Vector Math

Being able to manipulate vectors is a cornerstone technique for Deep Learning.
See if you can write functions that perform the following operations:

def elementwise_multiplication(vec_a, vec_b)   def vector_sum(vec_a)
def elementwise_addition(vec_a, vec_b)   def vector_average(vec_a)

Then, see if you can use two of these methods to perform a dot product!

---

(3) `Perform a Weighted Sum of Inputs`

```
def neural_network(input, weights):
    pred = w_sum(input,weights)
    return pred

def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(a):
        output += (a[i] * b[i])
    return output
```

```
                          local
   inputs   weights   predictions
 ( 8.50  *   0.1 )   =   0.85   = toes prediction
 ( 0.65  *   0.2 )   =   0.13   = wlrec prediction
 ( 1.20  *   0.0 )   =   0.00   = fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

       0.85      +      0.13      +      0.00      =      0.98
```

The intuition behind how and why a dot product (weighted sum) works is easily one of the most important parts of truly understanding how neural networks make predictions. Loosely stated, a dot product gives us a *notion of similarity* between two vectors. Consider the examples:

```
a = [ 0, 1, 0, 1]          w_sum(a,b)  = 0
b = [ 1, 0, 1, 0]          w_sum(b,c)  = 1
c = [ 0, 1, 1, 0]          w_sum(b,d)  = 1
d = [.5, 0,.5, 0]          w_sum(c,c)  = 2
e = [ 0, 1,-1, 0]          w_sum(d,d)  = .5
                           w_sum(c,e)  = 0
```

The highest weighted sum (w_sum(c,c)) is between vectors that are exactly identical. In contrast, since a and b have no overlapping weight, their dot product is zero. Perhaps the most interesting weighted sum is between c and e, since e has a negative weight. This negative weight cancelled out the positive similarity between them. However, a dot product between e and itself would yield the number 2, despite the negative weight (double negative turns positive). Let's become familiar with these properties.

Some have equated the properties of the "dot product" to a "logical AND". Consider a and b.

```
a = [ 0, 1, 0, 1]
b = [ 1, 0, 1, 0]
```

If you asked whether both a[0] AND b[0] had value, the answer would be no. If you asked whether both a[1] AND b[1] had value, the answer would again be no. Since this is AL-WAYS true for all 4 values, the final score equals 0. Each value failed the logical AND.

```
b = [ 1, 0, 1, 0]
c = [ 0, 1, 1, 0]
```

b and c, however, have one column that shares value. It passes the logical AND since b[2] AND c[2] have weight. This column (and only this column) causes the score to rise to 1.

```
c = [ 0, 1, 1, 0]
d = [.5, 0,.5, 0]
```

Fortunately, neural networks are also able to model partial ANDing. In this case, c and d share the same column as b and c, but since d only has 0.5 weight there, the final score is only 0.5. We exploit this property when modeling probabilities in neural networks.
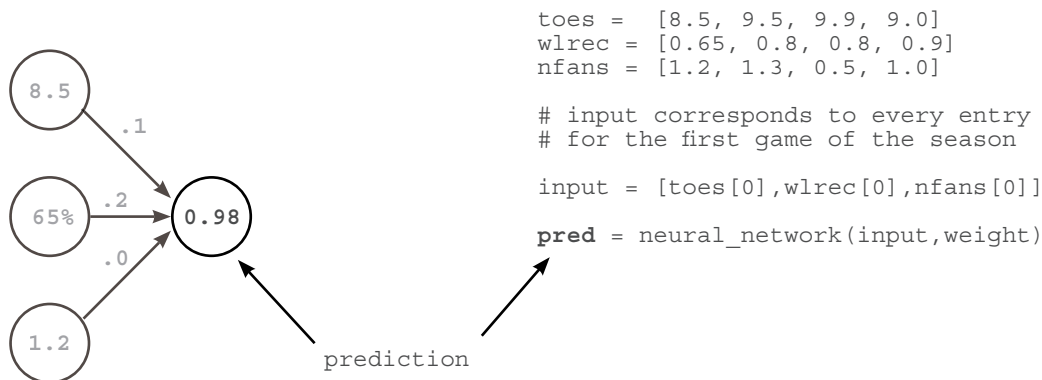
```
d = [.5, 0,.5, 0]
e = [-1, 1, 0, 0]
```

In this analogy, negative weights tend to imply a logcal NOT operator, given that any positive weight paired with a negative weight will cause the score to go down. Furthermore, if both vectors have negative weights (such as w_sum(e,e)), then it will perform a *double negative* and add weight instead. Additionally, some will say that it's an OR after the AND, since if any of the rows show weight, the score is affected. Thus, for w_sum(a,b), if (a[0] AND b[0]) OR (a[1] AND b[1])...etc.. then have a positive score. Furthermore, if one is negative, then that column gets a NOT. Amusingly, this actually gives us a kind of crude language to "read our weights". Let's "read" a few examples, shall we? These assume you're performing w_sum(input,weights) and the "then" to these "if statements" is just an abstract "then give high score".

```
weights = [ 1, 0, 1] => if input[0] OR input[2]

weights = [ 0, 0, 1] => if input[2]

weights = [ 1, 0, -1] => if input[0] OR NOT input[2]

weights = [ -1, 0, -1] => if NOT input[0] OR NOT input[2]

weights = [ 0.5, 0, 1] => if BIG input[0] or input[2]
```

Notice in the last row that a weight[0] = 0.5 means that the corresponding input [0] would have to be larger to compensate for the smaller weighting. And as I mentioned, this is a very *very* crude approximate language. However, I find it to be immensely useful when trying to picture in my head what's going on under the hood. This will help us significantly in the future, especially when putting networks together in increasingly complex ways.

So, given these intuitions, what does this mean when our neural network makes a prediction? Very rougly speaking, it means that our network gives a high score of our inputs based on *how similar they are to our weights*. Notice below that "nfans" is completely ignored in the prediction because the weight associated with it is a 0. The most sensitive predictor, in fact, is "wlrec" because its weight is a 0.2. However, the dominant force in the high score is the number of toes ("ntoes") not because the weight is the highest, but because the input combined with the weight is by far the highest.

---

④ `Deposit Prediction`


```
toes  = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

A few more points that we will note here for further reference. We cannot shuffle our weights. They have specific positions they need to be in. Furthermore, both the value of the weight AND the value of the input determine the overall impact on the final score. Finally, a negative weight would cause some inputs to reduce the final prediction (and vise versa).

Multiple Inputs - Complete Runnable Code

The code snippets from this example come together as follows.

Previous Code

We can create and execute our neural network using the following code. For the purposes of clarity, I have written everything out using only basic properties of Python (lists and numbers). However, there is a better way that we will start using in the future.

There is a python library called "numpy" which stands for "numerical python". It has very efficient code for creating vectors and performing common functions (such as a dot product). So, without further ado, here's the same code in numpy.

```python
def w_sum(a,b):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
        output += (a[i] * b[i])

    return output

weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

Numpy Code

```python
import numpy as np

weights = np.array([0.1, 0.2, 0])

def neural_network(input, weights):

    pred = input.dot(weights)

    return pred
```

```python
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]
pred = neural_network(input,weight)
print(pred)
```

```python
toes =  np.array([8.5, 9.5, 9.9, 9.0])
wlrec = np.array([0.65, 0.8, 0.8, 0.9])
nfans = np.array([1.2, 1.3, 0.5, 1.0])

# input corresponds to every entry
# for the first game of the season

input = np.array([toes[0],wlrec[0],nfans[0]])
pred = neural_network(input,weight)
print(pred)
```

**Both networks should
simply print out:**

0.98

Notice that we didn't have to create a special "w_sum" function. Instead, numpy has a special function called "dot" (short for "dot product") which we can call. Many of the functions we want to use in the future will have numpy parallels, as we will see later.

Making a Prediction with Multiple Outputs

Neural Networks can also make multiple predictions using only a single input.

Perhaps a simpler augmentation than multiple inputs is multiple outputs. Prediction occurs in the same way as if there were 3 disconnected single-weight neural networks.

(1) `An Empty Network With Multiple Outputs`
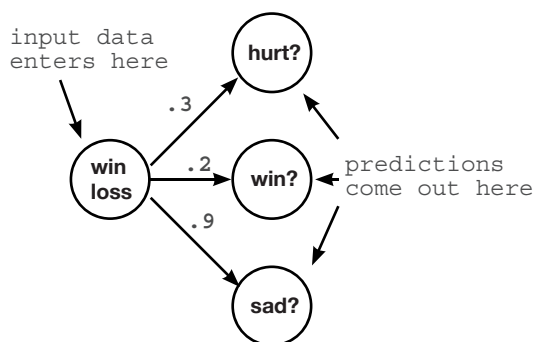

```
/* instead of predicting just
whether the team won or lost,
now we're also predicting whether
they are happy/sad AND the percentage
of the team that is hurt. We are
making this prediction using only
the current win/loss record */

weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred
```

The most important commentary in this setting is to notice that the 3 predictions really are completely separate. Unlike neural networks with multiple inputs and a single output where the prediction is undeniably connected this network truly behaves as 3 independent components, each receiving the same input data. This makes the network quite trivial to implement.
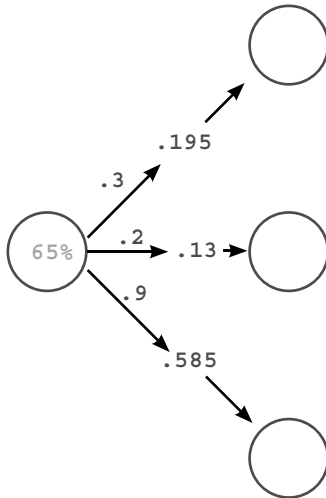
---

(2) `Inserting One Input Datapoint`
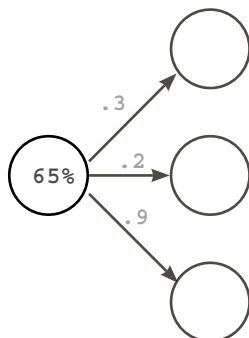

```
wlrec = [0.65, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weight)
```

---

③ Perform an Elementwise Multiplication

```
def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred

def ele_mul(number,vector):

    output = [0,0,0]

    assert(len(output) == len(vector))

    for i in xrange(len(vector)):
        output[i] = number * vector[i]

    return output
```

```
                         final
  inputs   weights   predictions
( 0.65  *  0.3 )  =  0.195  = hurt prediction
( 0.65  *  0.2 )  =  0.13   = win prediction
( 0.65  *  0.9 )  =  0.585  = sad prediction
```

④ Deposit Predictions

predictions
(a vector of numbers)
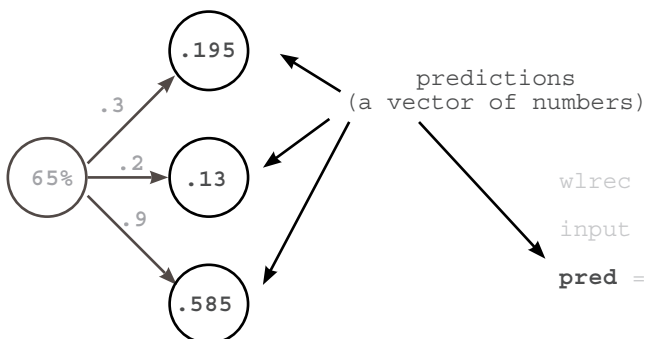
```
wlrec = [0.65, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weight)
```

Predicting with Multiple Inputs & Outputs

Neural networks can predict multiple outputs given multiple inputs.

Finally, the way in which we built a network with multiple inputs or outputs can be combined together to build a network that has both multiple inputs AND multiple outputs. Just like before, we simply have a weight connecting each input node to each output node and prediction occurs in the usual way.

---

① `An Empty Network With Multiple Inputs & Outputs`
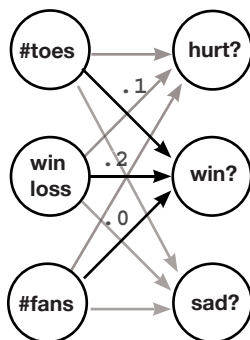
inputs   predictions


```
                 #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

---

② `Inserting One Input Datapoint`

inputs   predictions


```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

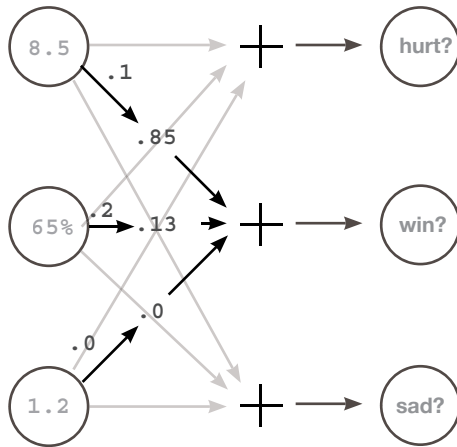**3** **For Each Output, Perform a Weighted Sum of Inputs**

```
def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred


def vect_mat_mul(vect,matrix):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
        output += (a[i] * b[i])

    return output
```

```
   #toes           %win           #fans
(8.5 * 0.1) + (0.65 * 0.1) + (1.2 * -0.3) = 0.555 = hurt prediction
(8.5 * 0.1) + (0.65 * 0.2) + (1.2 * 0.0)  = 0.98  = win prediction
(8.5 * 0.0) + (0.65 * 1.3) + (1.2 * 0.1)  = 0.965 = sad prediction
```

4 **Deposit Predictions**

inputs   predictions

```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

Multiple Inputs & Outputs - How does it work?

It performs 3 independent weighted sums of the input to make 3 predictions.

      I find that there are 2 perspectives one can take on this architecture. You can either think of it as 3 weights coming out of each input node, or 3 weights going into each output node. For now, I find the latter to be much more beneficial. For now, think about this neural network as 3 independent dot products, 3 independent weighted sums of the input. Each output node takes its own weighted sum of the input and makes a prediction.

---

① `An Empty Network With Multiple Inputs & Outputs`


```
                      #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

---

② `Inserting One Input Datapoint`


```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

---

③ **For Each Output, Perform a Weighted Sum of Inputs**
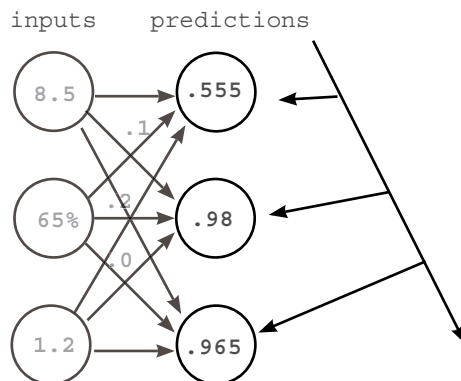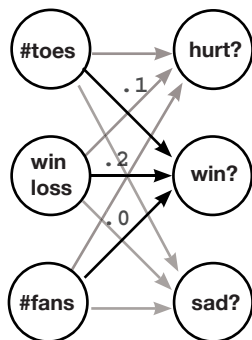


```
def neural_network(input, weights):
    pred = vect_mat_mul(input,weights)
    return pred

def vect_mat_mul(vect,matrix):
    assert(len(a) == len(b))
    output = vector_of_zeros(len(vect))
    for i in range(len(vect)):
        output[i] = w_sum(vect,matrix[i])
    return output
```

```
    #toes          %win          #fans
(8.5 * 0.1) + (0.65 * 0.1) + (1.2 * -0.3) = 0.555 = hurt prediction
(8.5 * 0.1) + (0.65 * 0.2) + (1.2 * 0.0)  = 0.98  = win prediction
(8.5 * 0.0) + (0.65 * 1.3) + (1.2 * 0.1)  = 0.965 = sad prediction
```
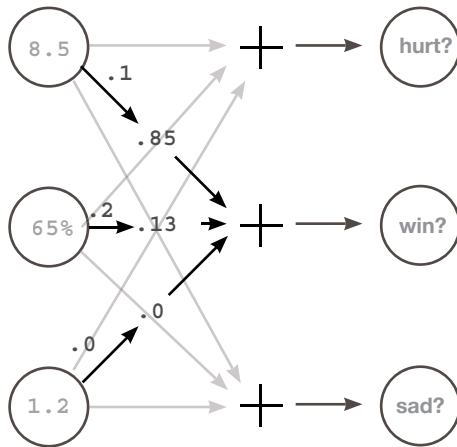
As mentioned on the previous page, we are choosing to think about this network as a series of weighted sums. Thus, in the code above, we created a new function called "vect_mat_mul". This function iterates through each row of our weights (each row is a vector), and makes a prediction using our w_sum function. It is literally performing 3 consecutive weighted sums and then storing their predictions in a vector called "output". There's a lot more weights flying around in this one, but isn't that much more advanced than networks we have previously seen.

I want to use this "list of vectors" and "series of weighted sums" logic to introduce you to two new concepts. See the weights variable in step (1)? It's a list of vectors. A list of vectors is simply called a **matrix**. It is as simple as it sounds. Furthermore, there are functions that we will find ourselves commonly using that leverage matrices. One of these is called **vector-matrix multiplication**. Our "series of weighted sums" is exactly that. We take a vector, and perform a dot product with every row in a matrix**. As we will find out on the next page, we even have special numpy functions to help us out.
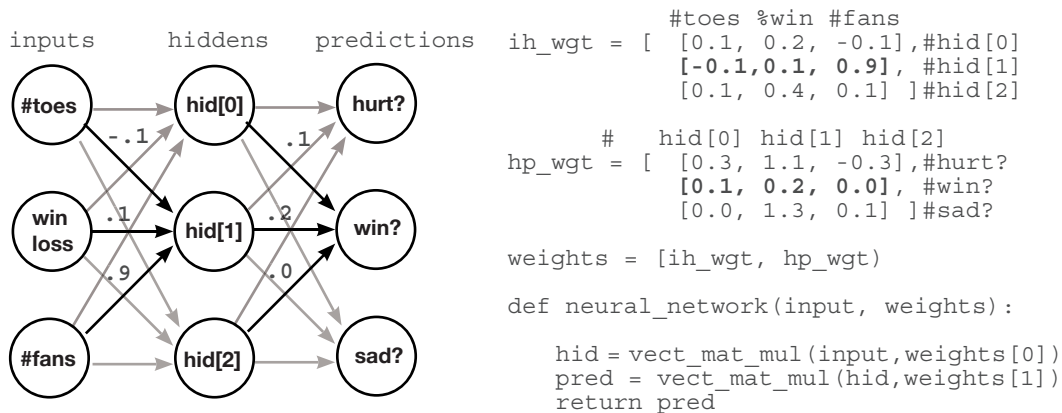

** Note: For those of you experienced with Linear Algebra, the more formal definition would store/process weights as column vectors instead of row vectors. This will be rectified shortly.

Predicting on Predictions

Neural networks can be stacked!

As the pictures below make clear, one can also take the output of one network and feed it as input to another network. This results in two consecutive vector-matrix multiplications. It may not yet be clear why you would predict in this way. However, some datasets (such as image classification) contain patterns that are simply too complex for a single weight matrix. Later, we will discuss the nature of these patterns. For now, it is sufficient that you know this is possible.

① **An Empty Network With Multiple Inputs & Outputs**


```
                       #toes %win #fans
ih_wgt = [ [0.1, 0.2, -0.1],#hid[0]
           [-0.1,0.1, 0.9], #hid[1]
           [0.1, 0.4, 0.1] ]#hid[2]

         #  hid[0] hid[1] hid[2]
hp_wgt = [ [0.3, 1.1, -0.3],#hurt?
           [0.1, 0.2, 0.0], #win?
           [0.0, 1.3, 0.1] ]#sad?

weights = [ih_wgt, hp_wgt)

def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```

② **Predicting the Hidden Layer**


```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)


def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```
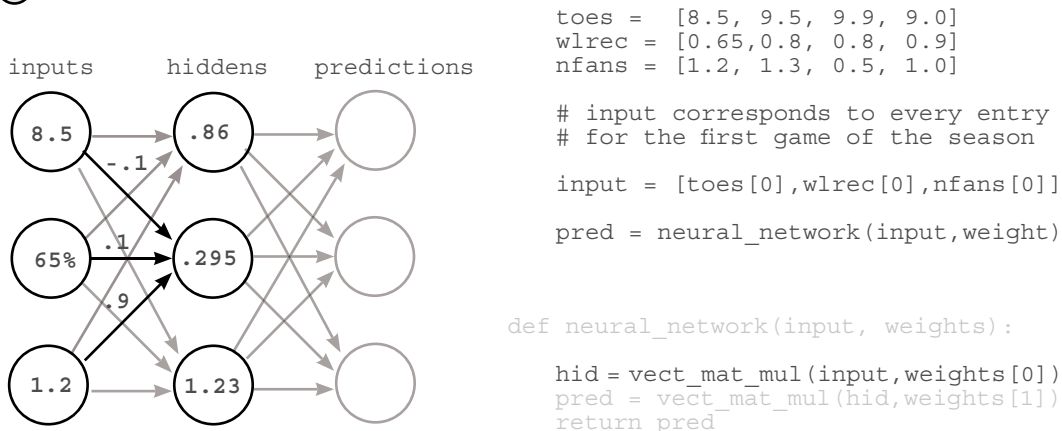
(3) **Predicting the Output Layer (and depositing the prediction)**

inputs   hiddens   predictions


```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)


def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```
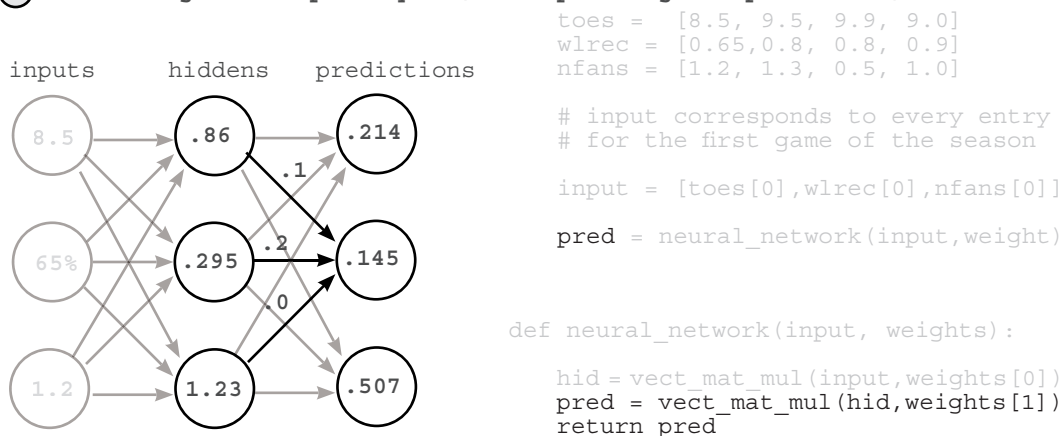
Numpy Version

```
import numpy as np

#toes %win #fans
ih_wgt = np.array([
          [0.1, 0.2, -0.1],#hid[0]
          [-0.1,0.1, 0.9], #hid[1]
          [0.1, 0.4, 0.1]]).T #hid[2]


# hid[0] hid[1] hid[2]
hp_wgt = np.array([
          [0.3, 1.1, -0.3],#hurt?
          [0.1, 0.2, 0.0], #win?
          [0.0, 1.3, 0.1] ]).T#sad?

weights = [ih_wgt, hp_wgt]

def neural_network(input, weights):

    hid = input.dot(weights[0])
    pred = hid.dot(weights[1])
    return pred


toes =  np.array([8.5, 9.5, 9.9, 9.0])
wlrec = np.array([0.65,0.8, 0.8, 0.9])
nfans = np.array([1.2, 1.3, 0.5, 1.0])

input = np.array([toes[0],wlrec[0],nfans[0]])

pred = neural_network(input,weights)
print pred
```

A Quick Primer on Numpy

Numpy is so easy to use that it does a few things for you. Let's reveal the magic.

So far in this chapter, we've discussed two new types of mathematical tools, vectors and matrices. Furthermore, we have learned about different operations that occur on vectors and matrices including dot products, elementwise multiplication and addition, as well as vector-matrix multiplication. For these operations, we've written our own python functions that can operate on simple python "list" objects. In the short term, we will keep writing/using these functions so that we make sure we fully understand what's going on inside them. However, now that we've mentioned both "numpy" and several of the big operations, I'd like to give you a quick run-down of basic "numpy" use so that you will be ready for our transition to "only numpy" a few chapters from now. So, let's just start with the basics again, vectors and matrices.

```
import numpy as np

a = np.array([0,1,2,3]) # a vector
b = np.array([4,5,6,7]) # another vector
c = np.array([[0,1,2,3],# a matrix
              [4,5,6,7]])

d = np.zeros((2,4))#(2x4 matrix of zeros)
e = np.random.rand(2,5) # random 2x5
# matrix with all numbers between 0 and 1

print a
print b
print c
print d
print e
```

```
                Output

[0 1 2 3]
[4 5 6 7]
[[0 1 2 3]
 [4 5 6 7]]
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
[[ 0.22717119  0.39712632
0.0627734   0.08431724
0.53469141]
 [ 0.09675954  0.99012254
0.45922775  0.3273326
0.28617742]]
```

We can create vectors and marices in multiple ways in numpy. Most of the common ones for neural networks are listed above. Note that the processes for creating a vector and a matrix are identical. If you create a matrix with only one row, you're creating a vector. Furthermore, as in mathematics in general, you create a matrix by listing (rows,columns). I say that only so that you can remember the order. Rows comes first. Columns comes second. Let's see some operations we can do on these vectors and matrices.

```
print a * 0.1 # multiplies every number in vector "a" by 0.1
print c * 0.2 # multiplies every number in matrix "c" by 0.2
print a * b # multiplies elementwise between a and b (columns paired up)
print a * b * 0.2 # elementwise multiplication then multiplied by 0.2
print a * c # since c has the same number of columns as a, this performs
# elementwise multiplication on every row of the matrix "c"

print a * e # since a and e don't have the same number of columns, this
# throws a "Value Error: operands could not be broadcast together with.."
```

Go ahead and run all of the code on the previous page. The first big of "at first confusing but eventually heavenly" magic should be visible on that page. When you multiply two variables with the "*" function, numpy automatically detects what kinds of variables you're working with and "tries" to figure out the operation you're talking about. This can be mega-convenient but sometimes makes numpy a bit hard to read. You have to make sure you keep up with what each variable type is in your head as you go along.

The general rule of thumb for anything elementwise (+,-,*,/) is that the two variables must either have the SAME number of columns, or one of the variables must only have 1 column.

For example, "print a * 0.1" takes a vector and multiplies it by a single number (a scalar). Numpy goes "oh, I bet I'm supposed to do vector-scalar multiplication here" and then it takes the scalar (0.1) and multiplies it by every value in the vector. This looks exactly the same as "print c * 0.2", except that numpy knows that c is a matrix. Thus, it performs scalar-matrix multiplication, multiplying every element in c by 0.2. Because the scalar has only one column, you can multiply it by anything (or divide, add, or subtract for that matter)

Next up, "print a * b". Numpy first identifies that they're both vectors. Since neither vector has only 1 column, it checks to see if they have an identical number of columns. Since they do, it knows to simply multiply each element by each element based on their positions in the vectors. The same is true with addition, subtraction and division.

"print a * c" is perhaps the most elusive. "a" is a vector with 4 columns. "c" is a (2x4) matrix. Neither have only one column, so numpy checks to see if they have the same number of columns. Since they do, numpy multiplies the vector "a" by each row of "c" (as if it was doing elementwise vector multiplication on each row).

Again, the most confusing part about this is that all of these operations look the same if you don't know which variables are scalars, vectors, or matrices. When I'm "reading numpy", I'm really doing 2 things, reading the operations and keeping track of the "shape" (number of rows and columns) of each operation. It'll take some practice, but eventually it becomes second nature.

```
a = np.zeros((1,4)) # vector of length 4
b = np.zeros((4,3)) # matrix with 4 rows & 3 columns

c = a.dot(b)
print c.shape
```

```
Output

(1,3)
```

There is one golden rule when using the 'dot' function. If you put the (rows,cols) description of the two variables you're "dotting" next to each other, neighboring numbers should always be the same. In this case, we're dot producting a (1,4) with a (4,3). Thus, it works fine, and outputs a (1,3). In terms of variable shape, you can think of it this way. Regardless of whether you're "dotting" vectors or matrices. Their "shape" (number of rows and columns) must line up. The columns on the "left" matrix must equal rows on the "right".

```
(a,b).dot(b,c) = (a,c)
```

```
a = np.zeros((2,4)) # matrix with 2 rows and 4 columns
b = np.zeros((4,3)) # matrix with 4 rows & 3 columns

c = a.dot(b)
print c.shape # outputs (2,3)


e = np.zeros((2,1)) # matrix with 2 rows and 1 columns
f = np.zeros((1,3)) # matrix with 1 row & 3 columns

g = e.dot(f)
print g.shape # outputs (2,3)
```

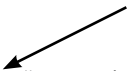**this ".T" "flips" the rows and columns of a matrix**

```
h = np.zeros((5,4)).T # matrix with 4 rows and 5 columns
i = np.zeros((5,6)) # matrix with 6 rows & 5 columns

j = h.dot(i)
print j.shape # outputs (4,6)


h = np.zeros((5,4)) # matrix with 5 rows and 4 columns
i = np.zeros((5,6)) # matrix with 5 rows & 6 columns
j = h.dot(i)
print j.shape # throws an error
```

Conclusion

To predict, neural networks perform repeated weighted sums of the input.

We have seen an increasingly complex variety of neural networks in this chapter. I hope that it is clear that a relatively small number of simple rules are simply used repeatedly to create larger, more advanced neural networks. Furthermore, the intelligence of the network really depends on what weight values we give to our networks.

In the next chapter, we will be learning how to set our weights so that our neural networks make accurate predictions. We will find that in the same way that prediction is actually based on several simple techniques that are simply repeated/stacked on top of each other, "weight learning" is also a series of simple techniques that are simply combined many times across an architecture. See you there!

Introduction to Neural Learning
Gradient Descent
4

IN THIS CHAPTER ·······························

“
The only relevant test of the validity of a hypothesis
is comparison of prediction with experience.
”

— *MILTON FRIEDMAN*

Predict, Compare, and Learn

This chapter is about "Compare", and "Learn"

In Chapter 3, we learned about the paradigm: "Predict, Compare, Learn". In the previous chapter, we dove deep into the first part of this process "Predict". In this process we learned a myriad of things including the major parts of neural networks (nodes and weights), how datasets fit into networks (matching the number of datapoints coming in at one time), and finally how to use a neural network to make a prediction. Perhaps this process begged the question, "How do we set our weight values so that our network predicts accurately?". Answering this question will be the main focus of this chapter, covering the second two steps of our paradigm, "Compare", and "Learn".

Compare

A measurement of how much our prediction "missed".

Once we've made a prediction, the next step to learn is to evaluate how well we did. Perhaps this might seem like a rather simple concept, but we will eventually find that coming up with a good way to measure error is one of the most important and complicated subjects of Deep Learning.

In fact, there are many properties of "measuring error" that you have likely been doing your whole life without realizing it. Perhaps you (or someone you know) amplifies bigger errors while ignoring very small ones. In this chapter we will learn how to mathematically teach our network to do this. Furthermore (and this might seem too simple to be important), we will learn that error is always positive! We will consider the analogy of an "archer" hitting a target. Whether he is too low by and inch or too high by an inch, the error is still just 1 inch! In our neural network "Compare" step, we want to consider these kinds of properties when measuring error.

As a heads up, in this chapter we will only evaluate one, very simple way of measuring error called "Mean Squared Error". However, it is but one of many ways to evaluate the accuracy of your neural network.

As a closing thought, this step will give us a sense for "how much we missed", but this isn't enough to be able to learn. The output of our "compare" logic will simply be a "hot or cold" type signal. Given some prediction, we'll calculate an error measure that will either say "a lot" or "a little". It won't tell us why we missed, what direction we missed, or what we should do to fix it. It more or less just says "big miss", "little miss", or "perfect prediction". What we do about our error is captured in the next step, "Learn".

Learn

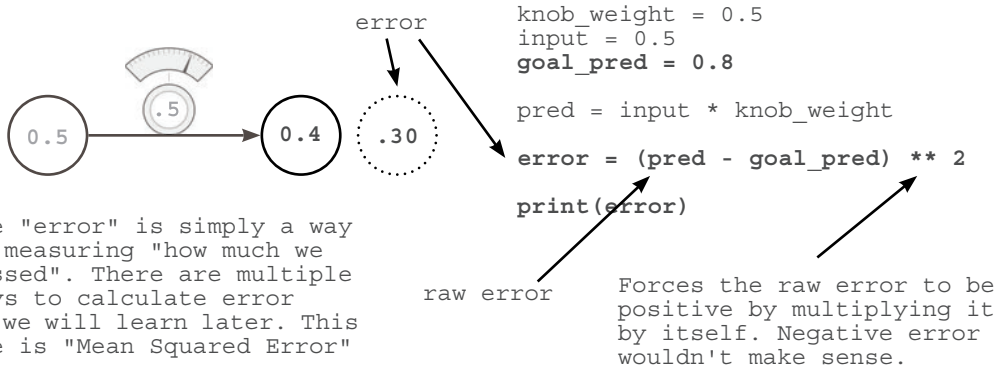**"Learning" takes our error and tells each weight how it can change to reduce it.**

Learning is all about "error attribution", or the art of figuring out how each weight played its part in creating error. It's the "blame game" of Deep Learning. In this chapter, we will spend a great number of pages learning the most popular version of the Deep Learning "blame game" called Gradient Descent.

At the end of the day, it's going to result in computing a number for each of our weights. That number will represent how that weight should be higher or lower in order to reduce the error. Then we will move the weight according to that number, and we'll be done.

Compare: Does our network make good predictions?

Let's measure the error and find out!

Execute this code in your Jupyter notebook. It should print "0.3025".

error

```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight

error = (pred - goal_pred) ** 2

print(error)
```

0.5 → 0.4   .30

The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error"

raw error

Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

What is the `goal_pred` variable?

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*. `pred-goal_pred` 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... 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's the Simplest Form of Neural Learning?

Learning using the Hot and Cold Method

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 is used 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 is predicting with perfect accuracy.

Hot and Cold Learning

Wiggling our weights to see which direction reduces the error the most, moving our weights in that direction, and repeating until the error gets to 0.
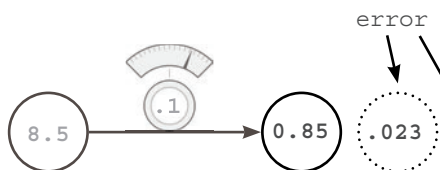
---

① **An Empty Network**

input data
enters here

predictions
come out here

.1

#toes → win?
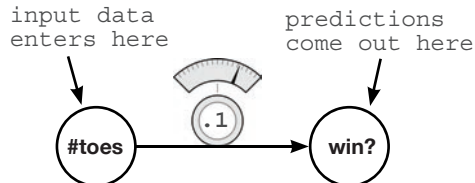
```
weight = 0.1

lr = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

---

② **PREDICT: Making A Prediction And Evaluating Error**

error

.1

8.5 → 0.85 .023

The "error" is simply a way
of measuring "how much we
missed". There are multiple
ways to calculate error
as we will learn later. This
one is "Mean Squared Error"

raw error

```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - true) ** 2
```
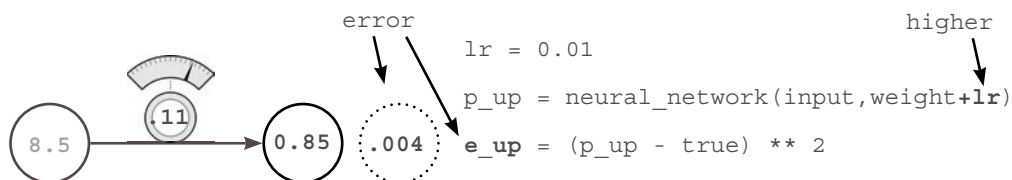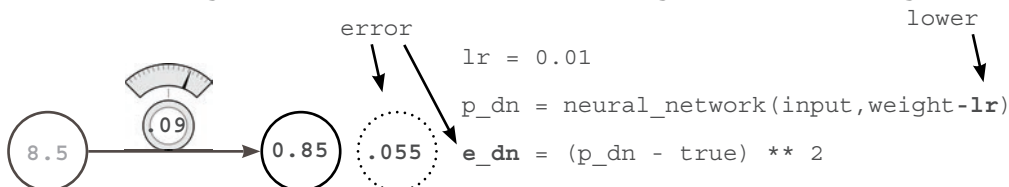
Forces the raw error to be
positive by multiplying it
by itself. Negative error
wouldn't make sense.

---

3 **COMPARE: Making A Prediction With a Higher Weight And Evaluating Error**

error   higher

```
lr = 0.01

p_up = neural_network(input,weight+lr)

e_up = (p_up - true) ** 2
```
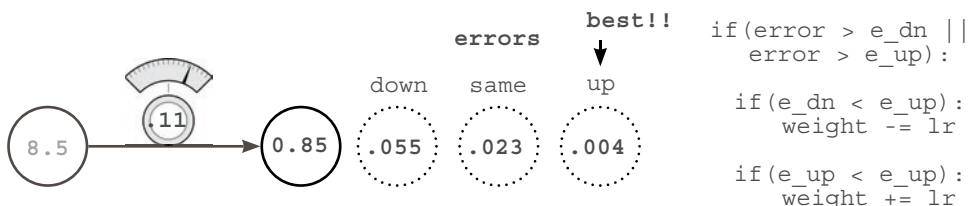
8.5   .11   0.85   .004

We want to move the weight so that the error goes downward, so we're
going to try moving the weight up and down to see which one has
the lowest error. First, we're trying moving the weight up (weight+lr).

4 **COMPARE: Making A Prediction With a Lower Weight And Evaluating Error**

error   lower

```
lr = 0.01

p_dn = neural_network(input,weight-lr)

e_dn = (p_dn - true) ** 2
```

8.5   .09   0.85   .055

5 **COMPARE + LEARN: Comparing our Errors and Setting our New Weight**

errors   **best!!**

down   same   up

```
if(error > e_dn ||
    error > e_up):

if(e_dn < e_up):
    weight -= lr

if(e_up < e_up):
    weight += lr
```

8.5   .11   0.85   .055   .023   .004

These last 5 steps comprise 1 iteration of Hot and Cold Learning. Fortunately, this iteration got us pretty close to the correct answer all by itself. (The new error is only 0.004). However, under normal circumstances, we would have to repeat this process many times in order to find the correct weights. Some people even have to train their networks for weeks or months before they find a good enough weight configuration.

This reveals what learning in neural networks really is. It's a **search problem**. We are *searching* for the best possible configuration of weights so that our network's error falls to zero (and predicts perfectly). As with all other forms of search, we might not find exactly what we're looking for, and even if we do, it may take some time. On the next page, we'll use Hot and Cold Learning for a *slightly* more difficult prediction so that you can see this searching in action!

Hot and Cold Learning

Perhaps the simplest form of learning.

Execute this code in your Jupyter Notebook. (New neural network modifications are in **bold**.)
<u>This code attempts to correctly predict 0.8.</u>

```
weight = 0.5                    how much to move
input = 0.5                     our weights each
goal_prediction = 0.8           iteration

step_amount = 0.001                 repeat learning many times
                                    so that our error can
for iteration in range(1101):       keep getting smaller

    prediction = input * weight                    TRY UP!
    error = (prediction - goal_prediction) ** 2

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

    up_prediction = input * (weight + step_amount)
    up_error = (goal_prediction - up_prediction) ** 2        TRY DOWN!

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

    if(down_error < up_error):                   If down is better,
        weight = weight - step_amount            go down!

    if(down_error > up_error):                   If up is better,
        weight = weight + step_amount            go up!
```

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

```
Error:0.3025 Prediction:0.25              Our last step correctly
Error:0.30195025 Prediction:0.2505        predicts 0.8!
         ....
Error:2.50000000033e-07 Prediction:0.7995
Error:1.07995057925e-27 Prediction: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 `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 #1: It's inefficient

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

PROBLEM #2: Sometimes it's impossible to predict the *exact* goal prediction.

With a set `step_amount`, unless the perfect `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!!!

The real problem here is that even though we know the correct **direction** to move our `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 our `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 `weight` value.

```
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...
```

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!

Execute this code in your Jupyter notebook.

direction_and_amount

(1) "pure error"

(2) scaling, negative reversal, and stopping

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print "Error:" + str(error) + " Prediction:" + str(pred)
```
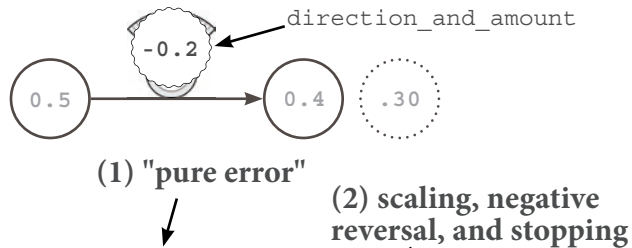
What you see above is a *superior* form of learning known as **Gradient Descent**. This method allows us to (in a single line of code... seen above in **bold**) calculate both the *direction* and the *amount* that we should change our `weight` so that we reduce our `error`.

What is the `direction_and_amount`?

It represents how we want to change our `weight`. The first **(1)** is what we call "pure error" which equals (`pred - goal_pred`). 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 `weight`.

What is the "pure error"?

It's the (`pred - goal_pred`) which indicates "the raw direction and amount that we missed". If this is a *positive* number then we predicted too *high* and vice 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 `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 `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 `weight` value has the same `error`... and moving it makes no difference because the `pred` is always 0.

What is "negative reversal"?

This is probably our most difficult and important effect. Normally (when `input` is positive), moving our `weight` *upward* makes our prediction move *upward*. However, if our `input` is *negative*, then all of a sudden our `weight` changes directions!!! When our `input` is *negative*, then moving our `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", ensuring that our `weight` moves in the correct direction, even if the `input` is negative.

What is "scaling"?

Scaling is the second effect 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* to address when this scaling goes out of control.

When you run the code in the top left, you should 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
```

Our last steps correctly approach 0.8!

In this example, we saw Gradient Descent in action in a bit of an oversimplified environment. On the next page, we're going to see it in it's more native environment. Some terminology will be different, but we will code it in a way that makes it more obviously applicable to other kinds of networks (such as those with multiple inputs and outputs)

One Iteration of Gradient Descent

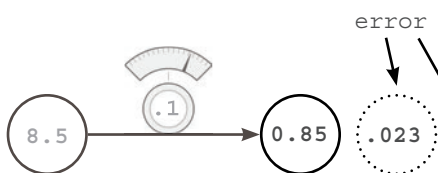**This performs a weight update on a single "training example" (input->true) pair**

---

(1) `An Empty Network`

input data
enters here

predictions
come out here

.1

#toes ──▶ **win?**
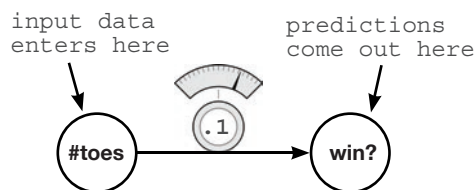
```
weight = 0.1

alpha = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

---

(2) `PREDICT: Making A Prediction And Evaluating Error`

error

.1

8.5 ──▶ 0.85   .023

The "error" is simply a way
of measuring "how much we
missed". There are multiple
ways to calculate error
as we will learn later. This
one is "Mean Squared Error"

```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)
```
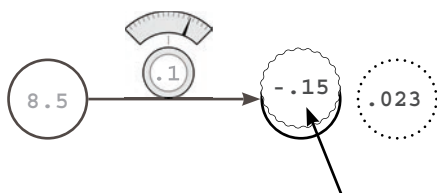
error = (pred - goal_pred) ** 2

raw error

Forces the raw error to be
positive by multiplying it
by itself. Negative error
wouldn't make sense.

(3) `COMPARE: Calculating "Node Delta" and Putting it on the Output Node`

.1

8.5 ──▶ -.15   .023

node delta

```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2
```
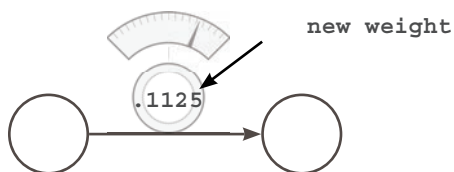
delta = pred - goal_pred

Delta is a measurement of "how much this node missed". Thus, since the
true prediction was 1.0, and our network's prediction was 0.85, the
network was too *low* by 0.15. Thus, delta is *negative* 0.15.

The primary difference between the gradient descent on the previous page and the implementation on this page just happened. `delta` is a new variable. It's the "raw amount that the node was too high or too low". Instead of computing `direction_and_amount` directly, we first calculate how much we wanted our output node to be different. Only then do we compute our `direction_and_amount` to change the `weight` (in step 4, now renamed "`weight_delta`").

---

④ **LEARN: Calculating "Weight Delta" and Putting it on the Weight**
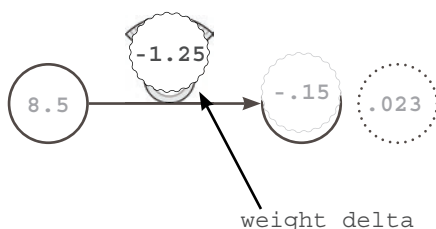
```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta
```

weight delta

Weight delta is a measure of "how much this weight caused the newtork to miss". We calculate it by multiplying the weight's output "Node Delta" by the weight's input. Thus, we create each "Weight Delta" by *scaling* it's output "Node Delta" by the weight's input. This accounts for the 3 aforementioned properties of our "direction_and_amount", scaling, negative reversal, and stopping.

---

⑤ **LEARN: Updating the Weight**

new weight

```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta

alpha = 0.01 // fixed before training

weight -= weight_delta * alpha
```

We multiply our weight_delta by a small number "alpha" before using it to update our weight. This allows us to control how fast the network learns. If it learns too fast, it can update weights too aggressively and overshoot. More on this later. Note that the weight update made the same change (small increase) as Hot and Cold Learning

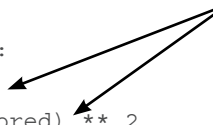# Learning Is Just Reducing Error

Modifying weight to reduce our error.

Putting together our code from the previous pages. We now have the following:

```
weight, goal_pred, input = (0.0, 0.8, 0.5)
                                              these lines have a secret
for iteration in range(4):

    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print "Error:" + str(error) + " Prediction:" + str(pred)
```

> **The Golden Method for Learning**
>
> Adjusting each `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 `weight` so that our `error` goes down. The secret to this lies in our `pred` and `error` calculations. Notice that we actually use our `pred` *inside* the `error` calculation. Let's replace our `pred` variable with the code we used to generate it.

```
error = ((input * weight) - goal_pred) ** 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 = ((0.5 * weight) - 0.8) ** 2
```

The Secret

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

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

Let's say that you moved `weight` up by 0.5... if there is an *exact relationship* between
`error` and `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?
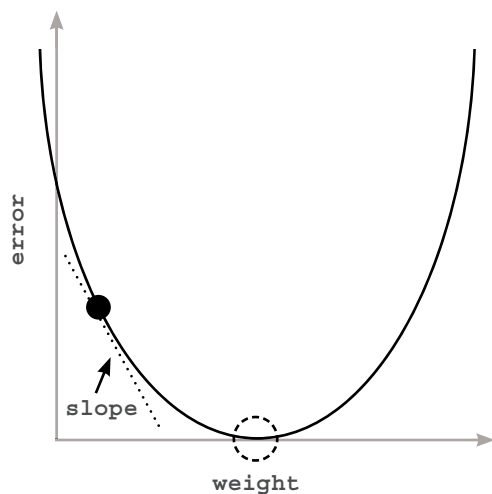
This graph represents *every value of error* for *every weight* according to the relationship in the
formula above. Notice it makes a nice *bowl shape*. The black "dot" is at the point of BOTH our
current `weight` and `error`. The dotted "circle" is where we want to be (`error == 0`).

Key Takeaway: The *slope* points to the <u>bottom</u> 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
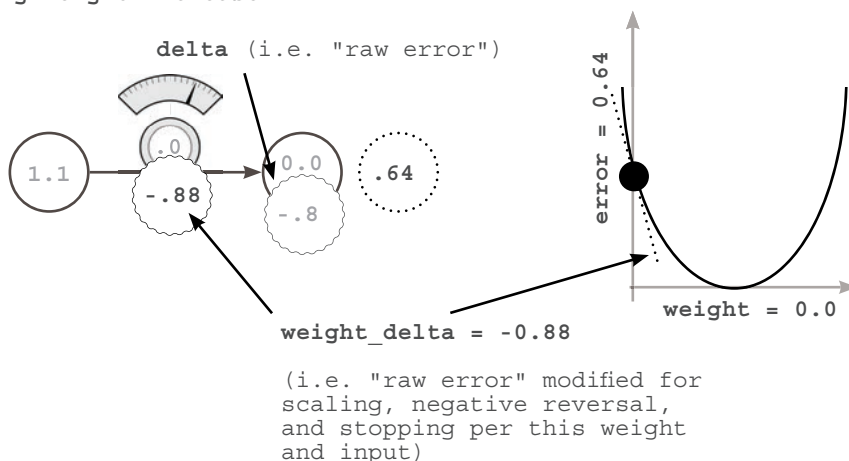error*.

Let's Watch Several Steps of Learning

Will we eventually find the bottom of the bowl?
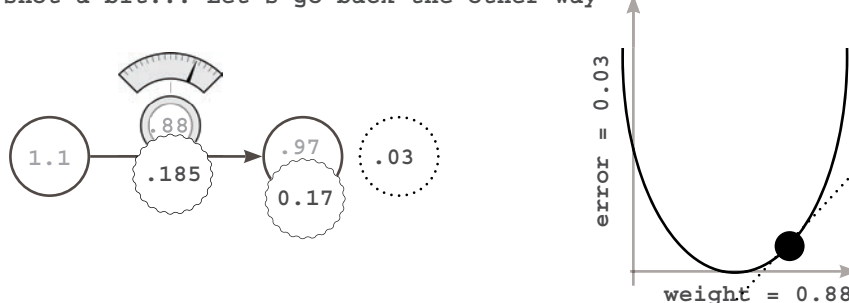
```
weight, goal_pred, input = (0.0, 0.8, 1.1)

for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print "Error:" + str(error) + " Prediction:" + str(pred)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```
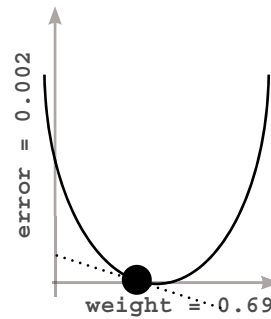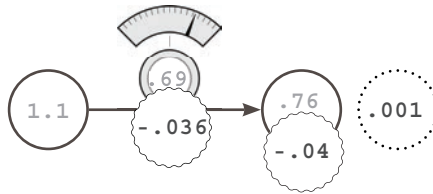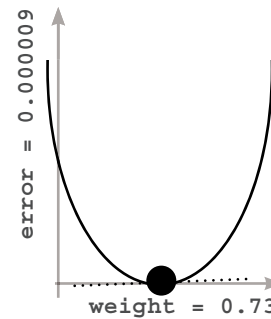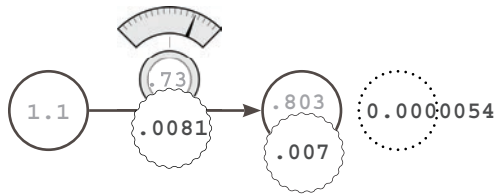
① **A Big Weight Increase**

delta (i.e. "raw error")

weight_delta = -0.88

(i.e. "raw error" modified for
scaling, negative reversal,
and stopping per this weight
and input)

② **Overshot a bit... Let's go back the other way**

③ Overshot Again! Let's go back again... but only just a little

④ Ok, we're pretty much there...

◯ Code Output

```
-----
Weight:0.0
Error:0.64 Prediction:0.0
Delta:-0.8 Weight Delta:-0.88
-----
Weight:0.88
Error:0.028224 Prediction:0.968
Delta:0.168 Weight Delta:0.1848
-----
Weight:0.6952
Error:0.0012446784 Prediction:0.76472
Delta:-0.03528 Weight Delta:-0.038808
-----
Weight:0.734008
Error:5.489031744e-05 Prediction:0.8074088
Delta:0.0074088 Weight Delta:0.00814968
```

Why does this work? What really is weight_delta?

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(s). 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 = ((input * weight) - goal_pred) ** 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_pred`, `input`, `**2`, `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_pred` 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 capability 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 (sidenote: this is loosely how "inceptionism works").

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 `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, changing weight 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 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).

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

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

We can modify *anything* in our `pred` 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 `pred` 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 `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 `weight`. We want to know the *direction* and the *amount* that the `error` changes when we change the `weight`. This is the goal. So far, we've used two different methods to attempt to understand this relationship.

You see, when we were "wiggling" our `weight` (hot and cold learning) 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 `weight` and our `error`. We just wiggled the `weight` up and down and watched for how it changed the `error`. Once we knew the relationship, we could move the `weight` in the right direction using two simple if statements.

```
if(down_error < up_error):
    weight = weight - step_amount

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

Now, let's go back to the formula from the previous pages, where we combined our pred and error logic. As mentioned, they quietly define an *exact relationship* between our `error` and our `weight`.

```
error = ((input * weight) - goal_pred) ** 2
```

This line of code, ladies and gentlemen, is the secret. This is a formula. This is the relationship between `error` and `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 `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 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** 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 red move when I tug on blue" 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

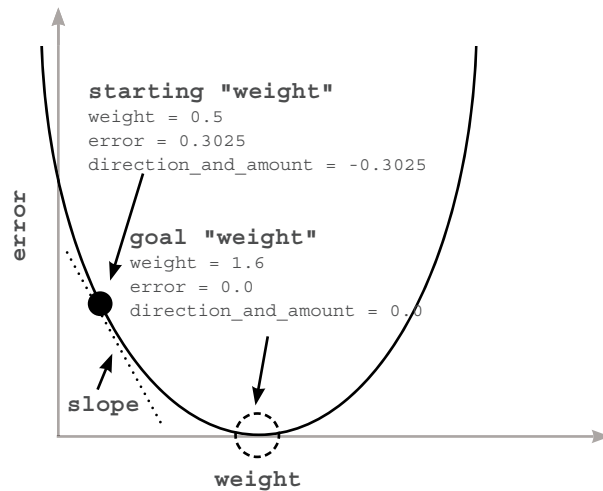**Still a little unsure about them?... let's take another perspective...**

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 = ((input * weight) - goal_pred) ** 2
```

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

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

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

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 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 *steepness* gives us **amount**. We can use both of these to help find the goal `weight`.


```
starting "weight"
weight = 0.5
error = 0.3025
direction_and_amount = -0.3025
```

```
goal "weight"
weight = 1.6
error = 0.0
direction_and_amount = 0.0
```

slope

error

weight

Even now, when I look at that curve, it's easy for me to lose track of what it represents. It's actually similar to our "hot and cold" method for learning. If we just tried *every possible value* for `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 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 `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 whopper of a 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 in a direction. Sorry to harp on, but it's important you know this in your bones.

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" sensity (when variables move together) and "negative" sensity (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. Ok, enough about derivatives. Let's get back to Gradient Descent.
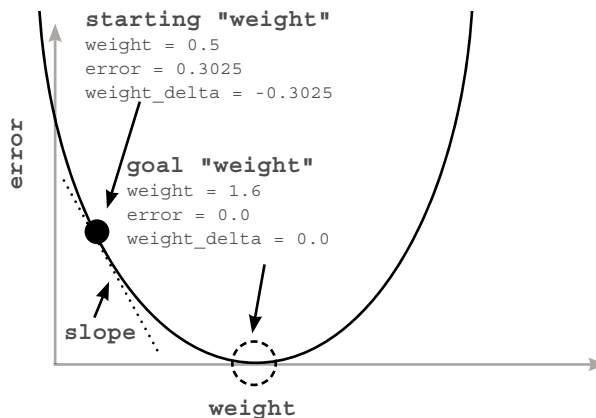
How to use a derivative to learn

"weight_delta" is our derivative.

What is the difference between the `error` and the derivative of our `error` and `weight`? Well the error is just a *measure* of how much we missed. The derivative defines the *realtionship* between each weight and how much we missed. In other words, it tells *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 `weight` starts out at (0.5). The dotted circle is where we want it to go... our goal `weight`. Do you see the dotted line attached to our black dot? That's our *slope* otherwise known as our *derivative*. It tells us *at that point in the curve* how much the `error` changes when we change the `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 `weight` to find the minimum of the `error`. Check it out!

```
starting "weight"
weight = 0.5
error = 0.3025
weight_delta = -0.3025

goal "weight"
weight = 1.6
error = 0.0
weight_delta = 0.0
```

slope

error

weight

So, how do we use our *derivative* to find the `error` minimum (lowest point in the `error` graph)? We just move 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 `weight` value *opposite the gradient* value, which *descends* our error to 0. By *opposite,* I simply mean that we increase our `weight` when we have a negative gradient and vice versa. It's like gravity!

Look Familiar?

```
weight = 0.0
goal_pred = 0.8
input = 1.1

for iteration in range(4):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta

    print "Error:" + str(error) + " Prediction:" + str(pred)
```
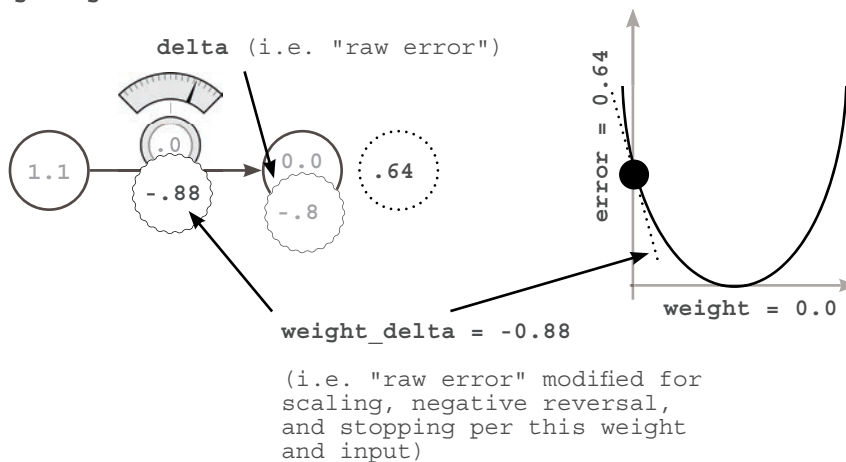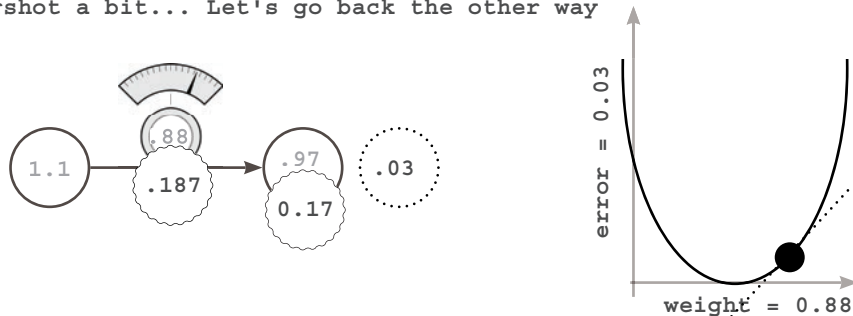
derivative
(i.e., how fast the
error changes given
changes in the weight)

---

① **A Big Weight Increase**

delta (i.e. "raw error")

1.1   .0   0.0   .64
      -.88   -.8

error = 0.64

weight = 0.0

weight_delta = -0.88

(i.e. "raw error" modified for
scaling, negative reversal,
and stopping per this weight
and input)

---

② **Overshot a bit... Let's go back the other way**

1.1   .88   .97   .03
      .187   0.17

error = 0.03

weight = 0.88

Breaking Gradient Descent

Just Give Me The Code

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = input * delta
    weight = weight - weight_delta
    print "Error:" + str(error) + " Prediction:" + str(pred)
```

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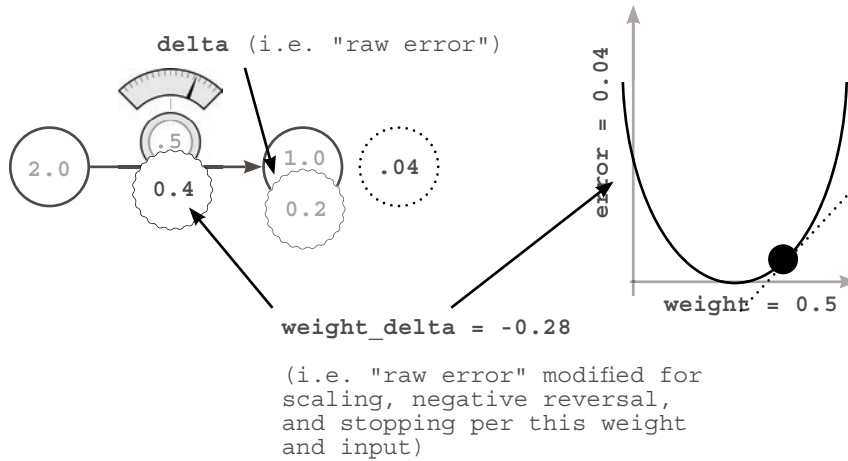
Let's try setting `input` to be equal to 2, but still try to get the algorithm to predict 0.8. What happens? Well, take a look at the output.

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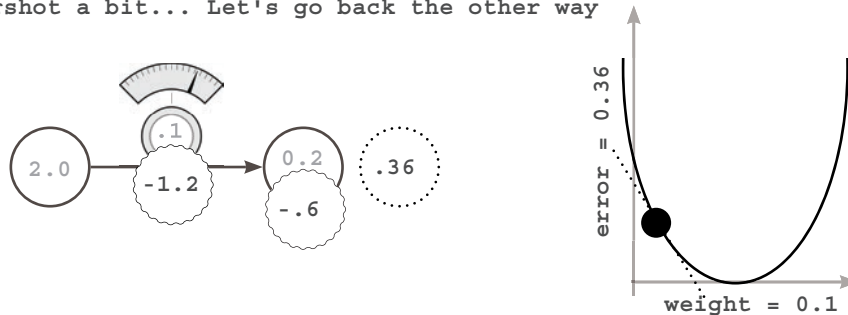
Woah! That's not what we wanted! Our predictions exploded! They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step! In other words, every update to our weight **overcorrects**! In the next section, we'll learn more about how to combat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

Visualizing the Overcorrections

① **A Big Weight Increase**

delta (i.e. "raw error")


```
2.0    .5    1.0    .04
       0.4   0.2
```

error = 0.04

weight = 0.5

weight_delta = -0.28

(i.e. "raw error" modified for
scaling, negative reversal,
and stopping per this weight
and input)

② **Overshot a bit... Let's go back the other way**

```
2.0    .1    0.2    .36
       -1.2  -.6
```

error = 0.36

weight = 0.1

③ **Overshot Again! Let's go back again... but only just a little**

```
2.0    1.3   2.6    3.24
       3.6   1.8
```

error = 3.24

weight = 1.3

Divergence

Sometimes... neural networks explode in value... oops?
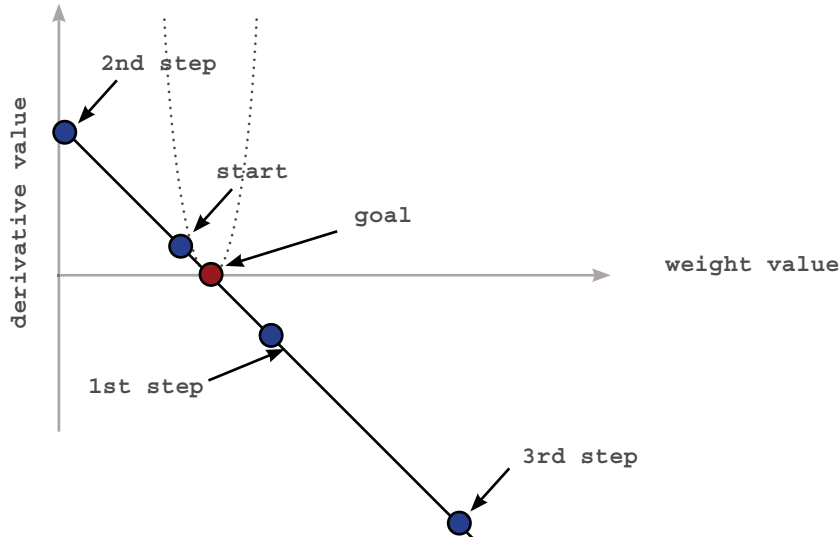
So what really happened? The explosion in error on the previous page is caused by the fact that we made the input larger. Consider how we're updating our weight.

```
weight = weight - (input * (pred - goal_pred))
```

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 `pred` = `input` * `weight`). This can cause our network to overcorrect. In other words, even though our `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 `weight`. So, if our input is *huge*, then small changes in our `weight` are going to cause BIG changes in our prediction!! The error is very *sensitive* to our `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.
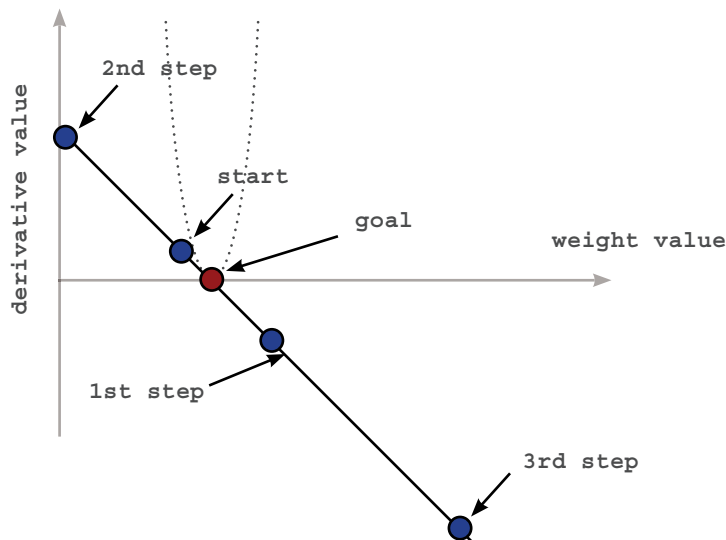
So, what was the problem we're trying to solve? The problem is this: if the input is too big, then our weight update can overcorrect. What is the symptom? The symptom is that when we overcorrect, our new derivative is even *larger in **magnitude*** than when we started (although the sign will be the 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` over time. If it starts 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.

```
weight = 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.

```
weight = 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)

```
weight = 0.5
goal_pred = 0.8
input = 2
alpha = 0.1

for iteration in range(20):
    pred  = input * weight
    error = (pred - goal_pred) ** 2
    derivative = input * (pred - goal_pred)
    weight = weight - (alpha * derivative)

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

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 plenty 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.

Learning Multiple Weights at a Time
Generalizing Gradient Descent

5

IN THIS CHAPTER ·······························

> " You don't learn to walk by following rules. You learn by doing, and by falling over. "
>
> — *RICHARD BRANSON*

Gradient Descent Learning with Multiple Inputs

Gradient Descent Also Works with Multiple Inputs

In the last chapter, we learned how to use Gradient Descent to update a weight. In this chapter, we will more or less reveal how the same techniques can be used to update a network that contains multiple weights. Let's start by just jumping in the deep end, shall we? The following diagram lists out how a network with multiple inputs can learn!

(1) An Empty Network With Multiple Inputs
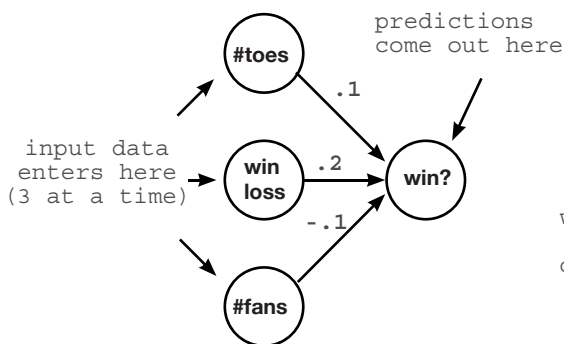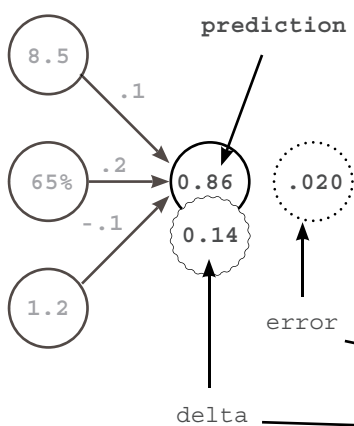


```
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(a):
        output += (a[i] * b[i])
    return output

weights = [0.1, 0.2, -.1]

def neural_network(input, weights):
    pred = w_sum(input,weights)
    return pred
```

(2) PREDICT+COMPARE: Making a Prediction and Calculating Error And Delta


```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.9, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

win_or_lose_binary = [1, 1, 0, 1]

true = win_or_lose_binary[0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)

error = (pred - true) ** 2

delta = pred - true
```
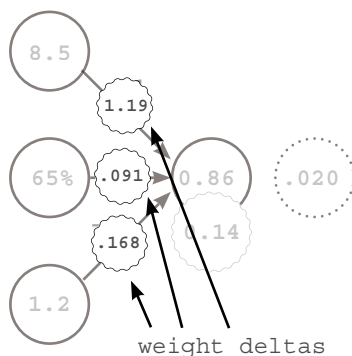
(3) **LEARN: Calculating Each "Weight Delta" and Putting It on Each Weight**

```
def ele_mul(number,vector):

    output = [0,0,0]

    assert(len(output) == len(vector))

    for i in xrange(len(vector)):
        output[i] = number * vector[i]

    return output


input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true

weight_deltas = ele_mul(delta,weights)
```
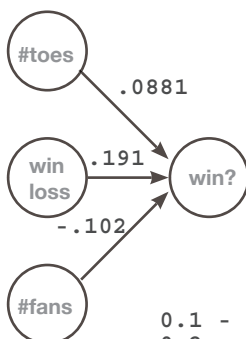
weight_deltas

There is actually nothing new in this diagram. Each weight delta is calculated by taking its output delta and multiplying it by its input. In this case, since our three weights share the same output node, they also share that node's delta. However, our weights have different weight deltas owing to their different input values. Notice further that we were able to re-use our ele_mul function from before as we are multiplying each value in weights by the same value delta.

```
8.5  *  0.14  =  1.19  = weight_deltas[0]
0.65 *  0.14  =  0.091 = weight_deltas[1]
1.2  *  0.14  =  0.168 = weight_deltas[2]
```
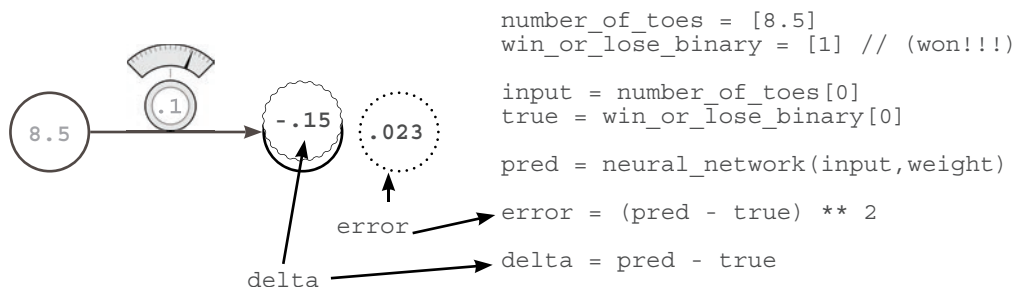
(4) **LEARN: Updating the Weights**

```
input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true

weight_deltas = ele_mul(delta,weights)

alpha = 0.01

for i in range(len(weights)):
    weights[i] -= alpha * weight_deltas[0]
```

#toes

.0881

win loss

.191

win?

-.102

#fans

```
 0.1 - (1.19 * 0.01) =  .0881 =  weights[0]
 0.2 - (.091 * 0.01) =  .191  =  weights[1]
-0.1 - (.168 * 0.01) = -.102  =  weights[2]
```

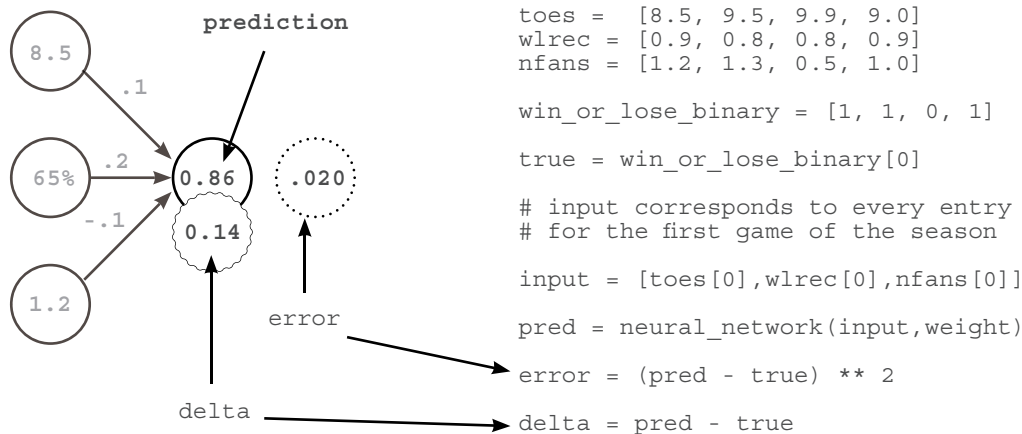# Gradient Descent with Multiple Inputs - Explained

Simple to execute, fascinating to understand.

When put side by side with our single-weight neural network, Gradient Descent with multiple inputs seems rather obvious in practice. However, the properties involved are quite fascinating and worthy of discussion. First, let's take a look at them side by side.

② Single Input: Making a Prediction and Calculating Error and Delta


```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true
```

② Multi Input: Making a Prediction and Calculating Error And Delta


```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.9, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

win_or_lose_binary = [1, 1, 0, 1]

true = win_or_lose_binary[0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true
```

Indeed, up until the generation of "delta" on the output node, single input and multi-input Stochastic Gradient Descent is identical (other than the prediction differences we studied in Chapter 3). We make a prediction, calculate the error and delta in the identical ways. However, the following problem remains, when we only had one weight, we only had one input (one weight_delta to generate). Now we have 3! How do we generate 3 weight_deltas?

How do we turn a single delta (on the node) into 3 weight_delta values?

Let's remember what the definition and purpose of delta is vs weight_delta. Delta is a measure of "how much we want a node's value to be different". In this case, we compute it by a direct subtraction between the node's value and what we wanted the node's value to be (pred - true). Positive delta indicates the node's value was too high, and negative that it was too low.

> **delta**
> A measure of how much we want a node's value to be higher or lower to predict "perfectly" given the current training example.

weight_delta, on the other hand, is an *estimate* for the direction and amount we should move our weights to reduce our node delta, inferred by the derivative. How do we transform our delta into a weight_delta? We multiply delta by a weight's input.
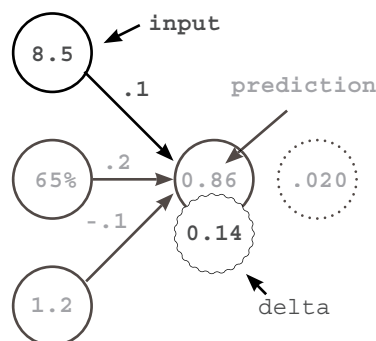
> **weight_delta**
> A derivative based *estimate* for the direction and amount we should move a weight to reduce our node_delta, accounting for scaling, negative reversal, and stopping.

Consider this from the perspective of a single weight, higlighted on the right. The delta says "Hey inputs! ... Yeah you 3!!! Next time, predict a little higher!". Then, our single weight says, "hmm, if my input was 0, then my weight wouldn't have mattered and i wouldn't change a thing (stopping). If my input was negative, then I'd want to decrease my weight instead of increase (negative reversal). However, my input is positive and quite large, so I'm *guessing* that my personal prediction mattered a lot to the aggregated output, so I'm going to move my weight up a lot to compensate! (Scaling)". It then increases it's weight.

So, what did those three properties/statements really say. They all three (stopping, negative reversal, and scaling) made an observation of how the weight's role in the delta was affected by its input! Thus, each weight_delta is a sortof "input modified" version of the delta.

Bringing us back to our original question, how do we turn one (node) delta into three weight_delta values? Well, since each weight has a unique input and a shared delta, we simply use each respective weight's input multiplied by the delta to create each respective weight_delta. It's really quite simple. Let's see this process in action on the next page.

Below you can see the generation of weight_delta variables for the previous single-input architecture and for our new multi-input architecture. Perhaps the easiest way to see how similar they are is by reading the psudocode at the bottom of each section. Notice that the multi-weight version (bottom of the page), simply multiplies the delta (0.14) by every input to create the various weight_deltas. It's really quite a simple process.

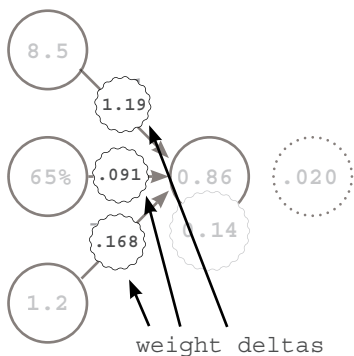(3) **Single Input: Calculating "Weight Delta" and Putting it on the Weight**
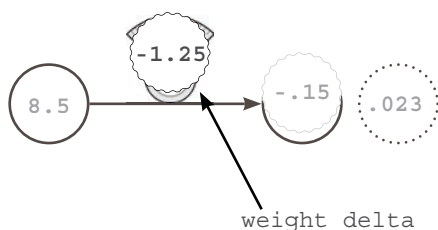
```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true

weight_delta = input * delta
```

weight delta

```
8.5  *  -0.15  =  -1.25  =>  weight_delta
```

---

(3) **Multi: Calculating Each "Weight Delta" and Putting It on Each Weight**
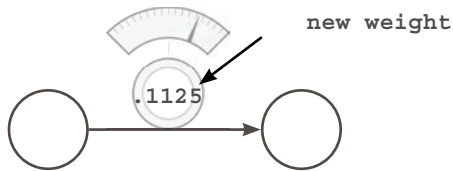
```
def ele_mul(number,vector):

    output = [0,0,0]

    assert(len(output) == len(vector))

    for i in xrange(len(vector)):
        output[i] = number * vector[i]

    return output


input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true

weight_deltas = ele_mul(delta,weights)
```

weight_deltas

```
8.5  *  0.14  =  1.19  =>  weight_deltas[0]
0.65 *  0.14  =  0.091 =>  weight_deltas[1]
1.2  *  0.14  =  0.168 =>  weight_deltas[2]
```

---

④ **Updating the Weight**

new weight

.1125

We multiply our weight_delta by
a small number "alpha" before
using it to update our weight.
This allows us to control how
fast the network learns. If it
learns too fast, it can update
weights too aggressively and
overshoot. More on this later.
Note that the weight update
made the same change (small
increase) as Hot and Cold
Learning

```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)


input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)


error = (pred - true) ** 2

delta = pred - true

weight_delta = input * delta


alpha = 0.01 // fixed before training

weight -= weight_delta * alpha
```
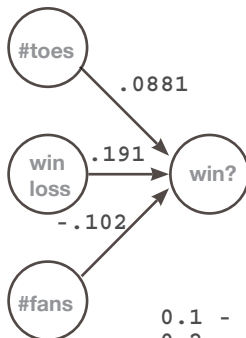
④ **Updating the Weights**


```
input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true

weight_deltas = ele_mul(delta,weights)

alpha = 0.01

for i in range(len(weights)):
    weights[i] -= alpha * weight_deltas[i]
```

```
0.1 - (1.19 * 0.01) =  .0881 =  weights[0]
0.2 - (.091 * 0.01) =  .191  =  weights[1]
-0.1 - (.168 * 0.01) = -.102  =  weights[2]
```

   Finally, the last step of our process is also nearly identical to the single-input network.
Once we have our weight_delta values, we simply multiply them by alpha and subtract them
from our weights. It's literally the same process as before, repeated across multiple weights in-
stead of just a single one.

Let's Watch Several Steps of Learning

```
def neural_network(input, weights):      for iter in range(3):
  out = 0
  for i in xrange(len(input)):             pred = neural_network(input,weights)
    out += (input[i] * weights[i])
  return out                               error = (pred - true) ** 2
                                           delta = pred - true
def ele_mul(scalar, vector):
  out = [0,0,0]                            weight_deltas=ele_mul(delta,input)
  for i in xrange(len(out)):
    out[i] = vector[i] * scalar            print "Iteration:" + str(iter+1)
  return out                               print "Pred:" + str(pred)
                                           print "Error:" + str(error)
toes  = [8.5, 9.5, 9.9, 9.0]               print "Delta:" + str(delta)
wlrec = [0.65, 0.8, 0.8, 0.9]              print "Weights:" + str(weights)
nfans = [1.2, 1.3, 0.5, 1.0]               print "Weight_Deltas:"
                                           print str(weight_deltas)
win_or_lose_binary = [1, 1,  0, 1]         print
true = win_or_lose_binary[0]
                                           for i in range(len(weights)):
alpha = 0.01                                 weights[i]-=alpha*weight_deltas[i]
weights = [0.1, 0.2, -.1]
input = [toes[0],wlrec[0],nfans[0]]
```
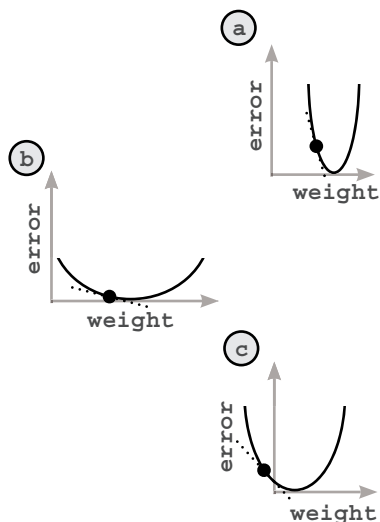

Notice that on the right, we can picture three individual error/weight curves, one for each weight. As before, the slopes of these curves (the dotted lines) are reflected by the "weight_delta" values. Furthermore, notice that (a) is steeper than the others. Why is the weight_delta steeper for (a) than the others if they share the same output delta and error measure? Well, (a) has an input value that is significantly higher than the others. Thus, a higher derivative.

2 **Iteration**

a

8.5

-.31 .112

b 65% -.02 .201 .964 .001

-.098 .04

-.04

1.2

c

weight_deltas

a

error

weight

b

error

weight

c

error

weight

3 **Iteration**

a

8.5

-.08 .115

b 65% -.01 .201 .991 .000

-.098 .01

-.01

1.2

c

weight_deltas

a

error

weight

b

error

weight

c

error
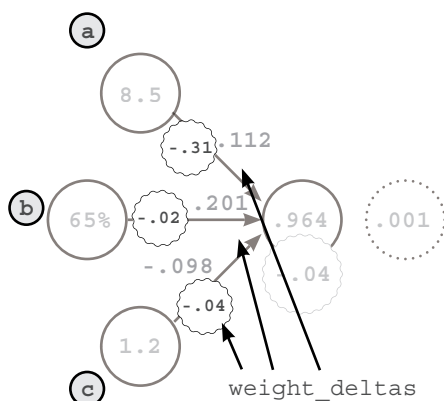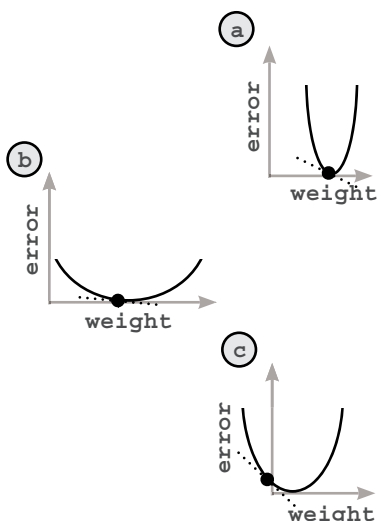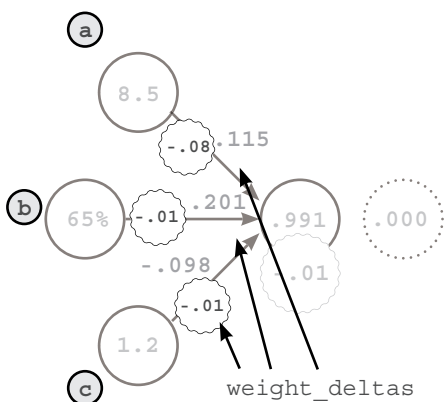
weight

A few additional takeaways: most of the learning (weight changing) was performed on the weight with the largest input (a), because the input changes the slope significantly. This isn't necessarily advantagoues in all settings. There is a sub-field called "normalization" that helps encourage learning across all weights despite dataset characteristics such as this. In fact, this significant difference in slope forced me to set the alpha to be lower than I wanted (0.01 instead of 0.1). Try setting alpha to 0.1. Do you see how (a) causes it to diverge?

Freezing One Weight - What Does It Do?

This experiment is perhaps a bit advanced in terms of theory, but I think that it's a great exercise to understand how the weights affect each other. We're going to train again, except weight a won't ever be adjusted. We'll try to learn the training example using only weights b and c (weights[1] and weights[2]).

```
def neural_network(input, weights):    for iter in range(3):
  out = 0
  for i in xrange(len(input)):           pred = neural_network(input,weights)
    out += (input[i] * weights[i])
  return out                             error = (pred - true) ** 2
                                         delta = pred - true
def ele_mul(scalar, vector):
  out = [0,0,0]                          weight_deltas=ele_mul(delta,input)
  for i in xrange(len(out)):             weight_deltas[0] = 0
    out[i] = vector[i] * scalar
  return out                             print "Iteration:" + str(iter+1)
                                         print "Pred:" + str(pred)
toes  = [8.5, 9.5, 9.9, 9.0]            print "Error:" + str(error)
wlrec = [0.65, 0.8, 0.8, 0.9]           print "Delta:" + str(delta)
nfans = [1.2, 1.3, 0.5, 1.0]            print "Weights:" + str(weights)
                                         print "Weight_Deltas:"
win_or_lose_binary = [1, 1, 0, 1]       print str(weight_deltas)
true = win_or_lose_binary[0]            print

alpha = 0.3                             for i in range(len(weights)):
weights = [0.1, 0.2, -.1]                weights[i]-=alpha*weight_deltas[i]
input = [toes[0],wlrec[0],nfans[0]]
```


① **Iteration**

weight_deltas

Perhaps you will be surprised to see that (a) still finds the bottom of the bowl? Why is this? Well, the curves are a measure of each individual weight relative to the global error. Thus, since the error is shared, when one weight finds the bottom of the bowl, all the weights find the bottom of the bowl.

This is actually an extremely important lesson. First of all, if we converged (reached error = 0) with (b) and (c) weights and then tried to train (a), (a) wouldn't move! Why? error = 0 which means weight_delt is 0! This reveals a potentailly damaging property of neural networks. (a) might be a really powerful input with lots of predictive power, but if the network accidentally figures how how to predict accurately on the training data without it, then it will never learn to incorporate (a) into its prediction.

Furthermore, notice "how" (a) finds the bottom of the bowl. Instead of the black dot moving, the curve seems to move to the left instead! What does this mean? Well, the black dot can only move horizontally if the weight is updated. Since the weight for (a) is frozen for this experiment, the dot must stay fixed. However, the error clearly goes to 0.

This tells us what the graphs really are. In truth, these are 2-d slices of a 4-dimensional shape. 3 of the dimensions are the weight values, and the 4th dimension is the error. This shape is called the "error plane" and, believe it or not, its curvature is determmiend by our training data! Why is it determined by our training data?

Well, our error is determined by our training data. Any network can have any weight value, but the value of the "error" given any particular weight configuration is 100% determined by data. We have already seen how the steepness of the "U" shape is affected by our input data (on several occasions). Truth be told, what we're really trying to do with our neural network is find the lowest point on this big "error plane", where the lowest point refers to the "lowest error". Interesting eh? We're going to come back to this idea later, so just file it away for now.
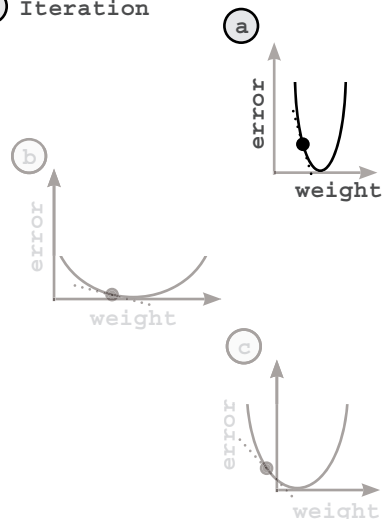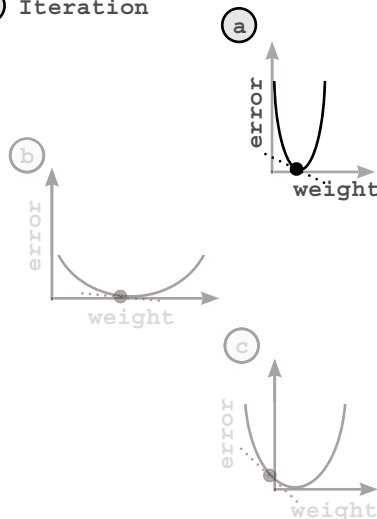
Gradient Descent Learning with Multiple Outputs

Neural Networks can also make multiple predictions using only a single input.

Perhaps this one will seem a bit obvious. We calculate each delta in the same way, and then multiply them all by the same, single input. This becomes each weight's weight_delta. At this point, I hope it is clear that a rather simple mechanism (Stochastic Gradient Descent) is consistently used to perform learning across a wide variety of architectures.

---

① **An Empty Network With Multiple Outputs**
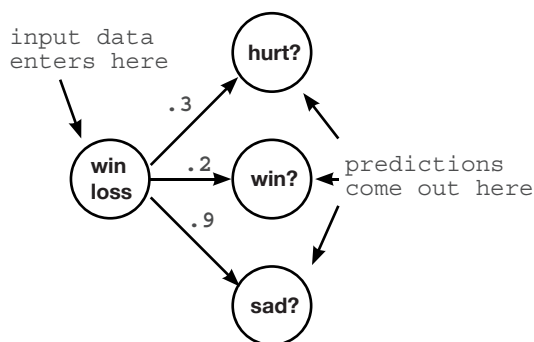
```
/* instead of predicting just
whether the team won or lost,
now we're also predicting whether
they are happy/sad AND the percentage
of the team that is hurt. We are
making this prediction using only
the current win/loss record */

weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred
```

---

② **PREDICT: Make a Prediction and Calculate Error and Delta**

```
wlrec = [0.9, 1.0, 1.0, 0.9]

hurt = [0.1, 0.0, 0.0, 0.1]
win  = [  1,   1,   0,   1]
sad  = [0.1, 0.0, 0.1, 0.2]

input = wlrec[0]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weight)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]
```

---

3 **COMPARE: Calculating Each "Weight Delta" and Putting It on Each Weight**

weight_deltas

.195   .009
.095
.062
65%   .492   .13   -.87
.153   .757
.585   .485
.235
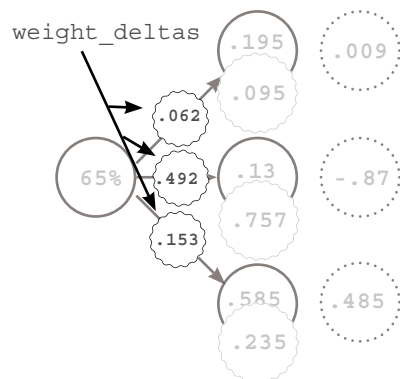
As before, weight_deltas are
computed by multiplying the
input node value with the
output node delta for each
weight. In this case, our
weight_deltas share the same
input node and have unique
output node (deltas). Note
also that we are able to
re-use our ele_mul function.

```
def ele_mul(number,vector):

    output = [0,0,0]

    assert(len(output) == len(vector))

    for i in xrange(len(vector)):
        output[i] = number * vector[i]

    return output

wlrec = [0.9, 1.0, 1.0, 0.9]

hurt  = [0.1, 0.0, 0.0, 0.1]
win   = [  1,   1,   0,   1]
sad   = [0.1, 0.0, 0.1, 0.2]

input = wlrec[0]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weight)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = ele_mul(input,weights)
```

4 **LEARN: Updating the Weights**
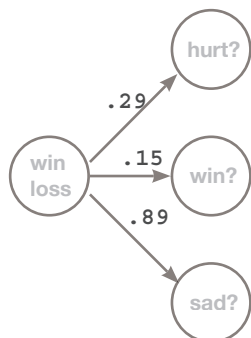
hurt?
.29
win
loss   .15   win?
.89
sad?

```
input = wlrec[0]
true = [hurt[0], win[0], sad[0]]
pred = neural_network(input,weight)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = ele_mul(input,weights)

alpha = 0.1

for i in range(len(weights)):
    weights[i] -= (weight_deltas[i] * alpha)
```

Gradient Descent with Multiple Inputs & Outputs

Gradient Descent generalizes to arbitrarily large networks.

① `An Empty Network With Multiple Inputs & Outputs`
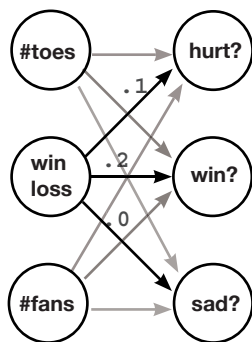
inputs   predictions


```
                    #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

② `PREDICT: Make a Prediction and Calculate Error and Delta`

inputs   pred   errors
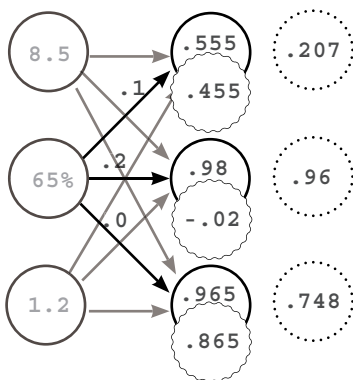

```
toes  = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

hurt  = [0.1, 0.0, 0.0, 0.1]
win   = [  1,   1,   0,   1]
sad   = [0.1, 0.0, 0.1, 0.2]

alpha = 0.01

input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weight)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]
```

③ **COMPARE: Calculating Each "Weight Delta" and Putting It on Each Weight**

inputs   pred   errors

```
8.5        .555      .207
       .296   .455
65%   -.01   .98      .96
            -.02
       .562
1.2        .965      .748
            .865
```

(weight deltas only
shown for one input
to save space)

```python
def outer_prod(vec_a, vec_b):

    out = zeros_matrix(len(a),len(b))

    for i in range(len(a)):
       for j in range(len(b)):
           out[i][j] = vec_a[i]*vec_b[j]

    return out

input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weight)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = outer_prod(input,delta)
```

④ **LEARN: Updating the Weights**

inputs   predictions

```
#toes          hurt?
        .09
win
loss    .2     win?
        01
#fans          sad?
```

```python
input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weight)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = outer_prod(input,delta)

for i in range(len(weights)):
  for j in range(len(weights[0])):
    weights[i][j] -= alpha * \
                 weight_deltas[i][j]
```
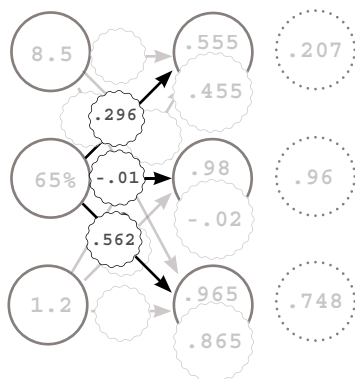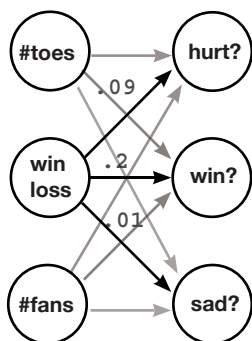
What do these weights learn?

Each weight tries to reduce the error, but what do they learn in aggregate?

Congratulations! This is the part of the book where we move onto our first *real world data-set*. As luck would have it, it's one with historical significance!

Our new dataset is called the MNIST dataset, which is a dataset comprised of digits that high school students and employees of the US Census bureau hand wrote some years ago. The interesting bit is that these handwritten digits are simply black and white images of people's handwriting. Accompanying each digit image is the actual number that they were writing (0-9). For the last few decades, people have been using this dataset to train neural networks to read human handwriting, and today, you're going to do the same!

Each image is only 784 pixels (28 x 28). So, given that we have 784 pixels as input and 10 possible labels as output, you can imagine the shape of our neural network. So, now that each training example contains 784 values (one for each pixel), our neural network must have 784 input values. Pretty simple, eh! We just adjust the number of input nodes to reflect how many data points are in each training example. Furthermore, we want to predict *10 probabilities*, one for each digit. In this way, given an input drawing, our neural network will produce these 10 probabilities, telling us which digit is most likely to be what was drawn.

So, how do we configure our neural network to produce ten probabilties? Well, on the last page, we saw a diagram for a neural network that could take multiple inputs at a time and make multiple predictions based on that input. Thus, we should be able to simply modify this network to have the correct number of inputs and outputs for our new MNIST task. We'll just tweak it to have 784 inputs and 10 outputs.

In the notebook entitled "<NOTEBOOK NAME>", you'll see a script to pre-process the MNIST dataset and load the first 1000 *images* and *labels* into two numpy matrices called `images` and `labels`. You may be wondering, "images are 2-dimensional... how do we load the (28 x 28) pixels into a flat neural network?" For now, the answer is quite simple. We "flatten" the images into a vector of 1 x 784. So, we take the first row of pixels and concatenate them with the second row, and third row, and so on until we have one long list of pixels per image (784 pixels long in fact).

inputs   predictions

pix[0] → 0?

pix[1] → 1?

pix[2] → 2?

pix[783] → 9?
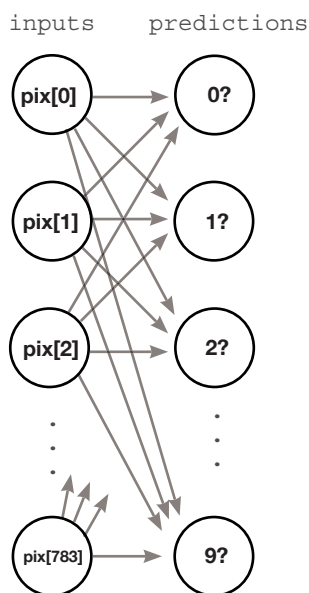
This picture on the left represents our new "MNIST Classification" neural network. It most closely resembles the network we trained with "Multiple Inputs and Outputs" a few pages ago. The only difference is the number of inputs and outputs, which has increased substantially. This network has 784 inputs (one for each pixel in a 28x28 image) and 10 outputs (one for each possible digit in the image).

If this network was able to predict perfectly, it would take in an image's pixels (say a 2 like the one on the previous page), and predict a 1.0 in the correct output position (the third one) and a 0 everywhere else). If it was able to do this correctly for all of the images in our dataset, it would have no error.

inputs   predictions

0.0 → 0.01

0.0 → 0.03

0.98 → 0.98

0.95 → 0.15

Highest Prediction! Thus the network thinks that this image is a "2"

Small Errors: This network thinks it kindof looks like a 9 (but only a bit)
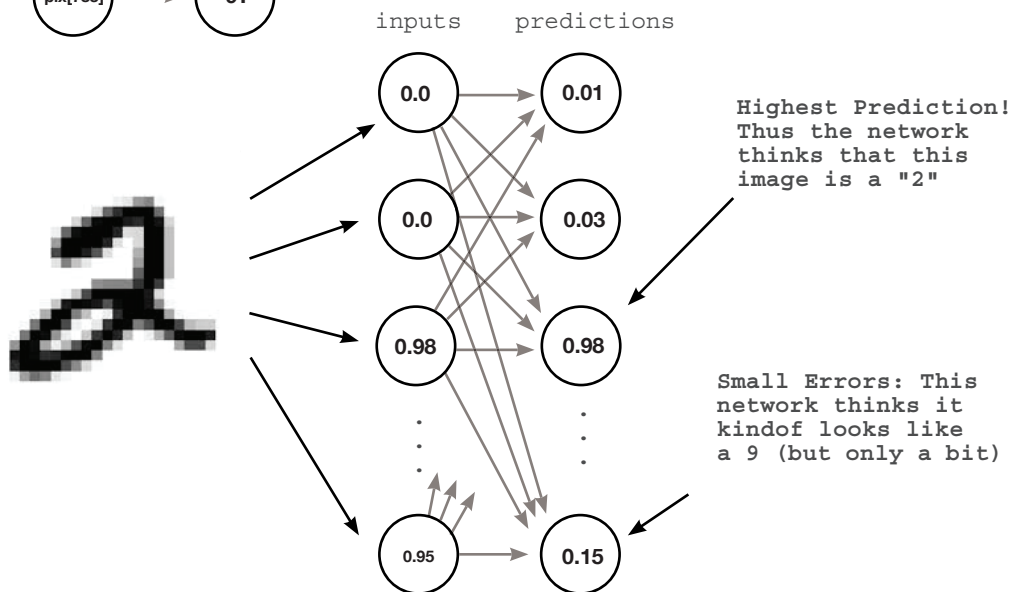
Over the course of training, the network will adjust the weights between the "input" and "prediction" nodes so that the error falls toward 0 in training. However, what does this actually do? What does it mean to modify a bunch of weights to learn a pattern in aggregate?

Visualizing Weight Values

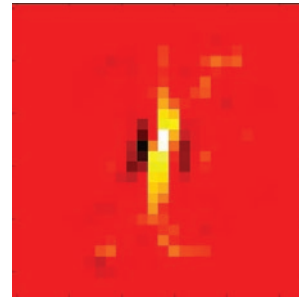### Each weight tries to reduce the error, but what do they learn in aggregate?

Perhaps an interesting and intuitive practice in neural network research (particularly for image classifiers) is to visualize the weights as if they were an image. If you look at the diagram on the right, you will see why.

Each output node has a weight coming from every pixel. For example, our "2?" node has 784 input weights, each mapping the relationship between a pixel and the number "2". What is this relationship? Well, if the weight is high, it means that the model believes there's a high degree of *correlation* between that pixel and the number 2. If the number is very low (negative), then the network believes there is a very low correlation (perhaps even negative correlation) between that pixel and the number two.

Thus, if we take our weights and print them out into an image that's the same shape as our input dataset images, we can "see" which

pixels have the highest correlation with a particular output node. As you can see above, there is a very vague "2" and "1" in our two images, which were created using the weights for "2" and "1" respectively. The "bright" areas are high weights, and the dark areas are negative weights. The neutral color (red if you're reading this in color) represents 0s in the weight matrix. This describes that our network generally knows the shape of a 2 and of a 1.

Why does it turn out this way? Well, this takes us back to our lesson on "dot products". Let's have a quick review, shall we?
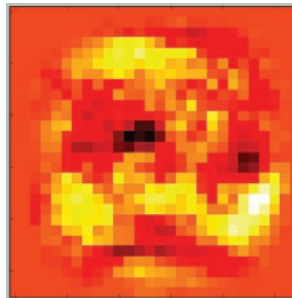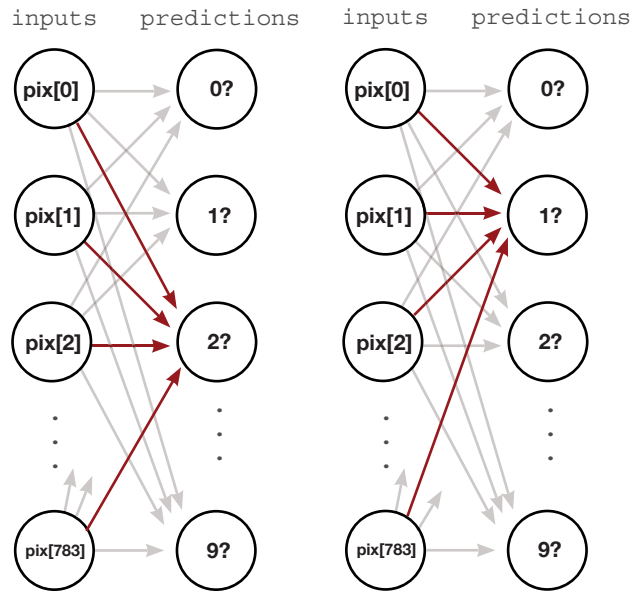
Visualizing Dot Products (weighted sums)

Each weight tries to reduce the error, but what do they learn in aggregate?

Recall how dot produts work. They take two vectors, multiply them together (elementwise), and then sum over the output. So, in the example below:

```
a = [ 0, 1, 0, 1]          score
b = [ 1, 0, 1, 0]

    [ 0, 0, 0, 0] -> 0
```

First, you would multiply each element in a and b by each other, in this case creating a vector of 0s. The sum of this vector is also zero. Why? Well, the vectors had nothing in common.

```
c = [ 0, 1, 1, 0]          b = [ 1, 0, 1, 0]
d = [.5, 0,.5, 0]          c = [ 0, 1, 1, 0]
```

However, dot products between c and d return higher scores, because there is overlap in the columns that have positive values. Furthermore, performing dot products between two identical vectors tend to result in higher scores as well. The takeaway? A **dot product is a loose measurement of similarity between two vectors.**

What does this mean for our weights and inputs? Well, if our weight vector is similar to our input vector for "2", then it's going to output a high score because the two vectors are similar! Inversely, if our weights vector is NOT similar to our input vector for 2, then it's going to output a low score. You can see this in action below! Why is the top score (0.98) higher than the lower one (0.01)?

inputs **(dot)** weights **(equals)** predictions

0.98

0.01
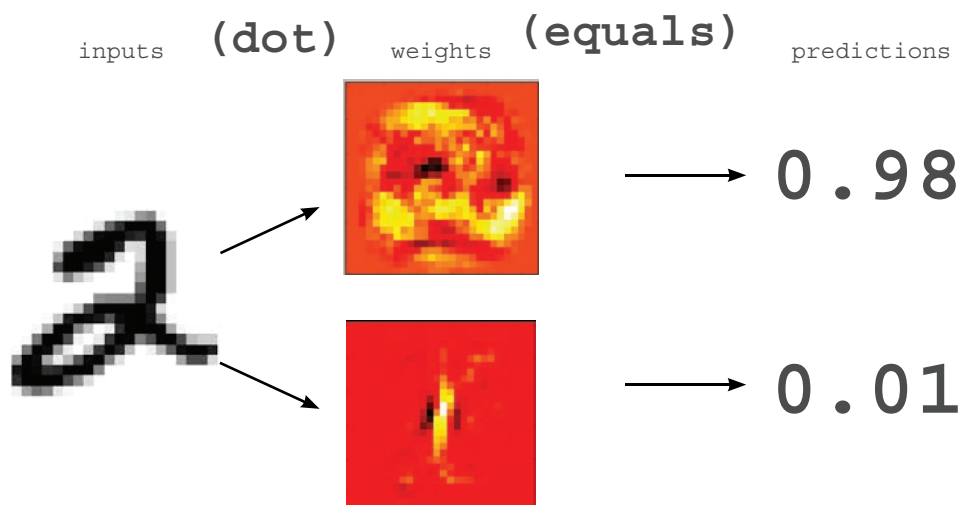
Conclusion

Gradient Descent is a General Learning Algorithm

Perhaps the most important subtext of this chapter is that Gradient Descent is a very flexible learning algorithm. If you combine weights together in a way that allows you to calculate an error function and a delta, gradient descent can show you how to move your weights to reduce your error. We will spend the rest of this book exploring different types of weight combinations and error functions for which Gradient Descent is useful. The next chapter is no exception.

Building Your First "Deep" Neural Network
Introduction to Backpropagation

6

IN THIS CHAPTER ·······································

> " Who invented backpropagation? "
>
> — *JURGEN SCHMIDHUBER*

The Street Light Problem

A toy problem for us to consider how a network learns entire datasets.

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?
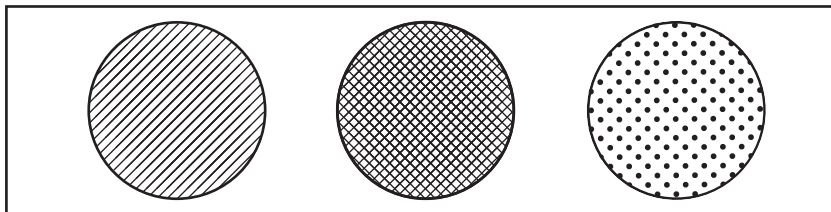
　　　　You can know when it is safe to cross the street by interpreting the streetlight. However, in this case, we don't know how to interpret it! Which "light" combinations indicate when it is time to **walk**? Which indicate when it is time to **stop**? To solve this problem, you might sit at the street corner 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.
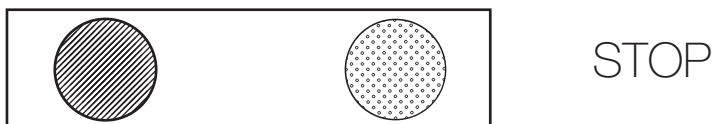
STOP

　　　　Ok, nobody walked at the first light. At this point you're 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.
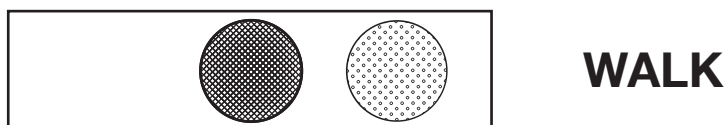
WALK

　　　　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.

STOP

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?
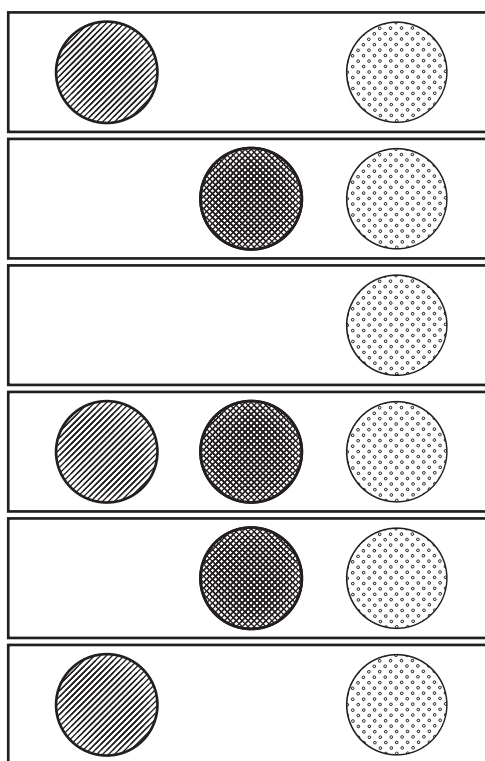
STOP

WALK

STOP

WALK

WALK

STOP

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.

What We Know	What We Want to Know
	STOP
	WALK
	STOP
	WALK
	WALK
	STOP

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.

Streetlights		Streetlight **Pattern**		
	→	1	0	1
	→	0	1	1
	→	0	0	1
	→	1	1	1
	→	0	1	1
	→	1	0	1

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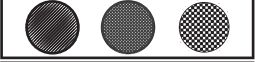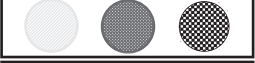
This structure of 1s and 0s is called a **matrix**. Furthermore, this relationship between the rows and columns is very common in matrices, especially matrices of data (like our streetlights).

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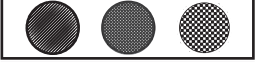
So, a column contains every state we recorded a thing in. In this case, a column contains every on/off state we recorded of a particular light. Each row contains the simultaneous state of every light at a particular moment in time. Again, this is common.

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**

.9	.0	1
.2	.8	1
.1	.0	1
.8	.9	1
.1	.7	1
.9	.1	0

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**

9	0	10
2	8	10
1	0	10
8	9	10
1	7	10
9	1	0

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
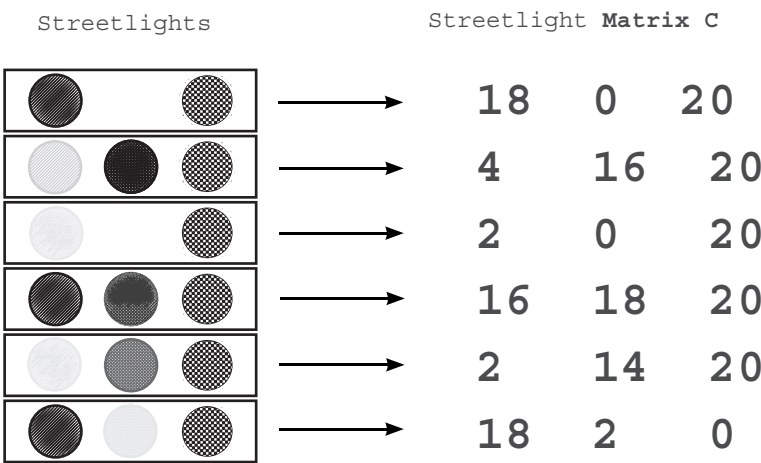
It's important to recognize that "the underlying pattern" is not the same as "the matrix". It's a property of the matrix. In fact, it's a property of all three of these matrices (A, B, and C). The pattern is what each of these matrices is *expressing.* The pattern also existed in the streetlights. This *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 decode the 1s and 0s into the underlying STOP/WALK pattern. We call this resulting matrix a **lossless representation** 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
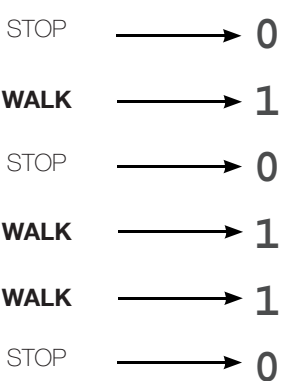
Creating a Matrix or Two in Python

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.

```python
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.

```python
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.

Neural Network

streetlights ⟶ [Neural Network] ⟶ walk _ vs _ stop
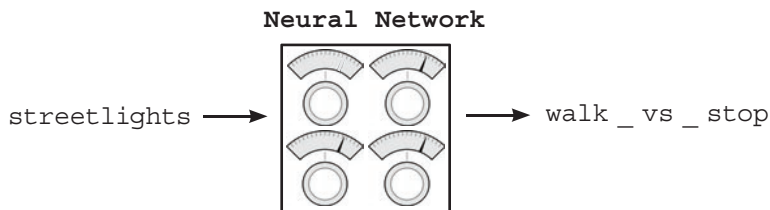
Building Our Neural Network

Ok, so we've been learning about neural networks for several chapters now. We've got a new dataset, and we're going to create a neural network to solve it. Below, I've written out some example code to learn the first streetlight pattern. This should look very familiar.

```python
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)
```

Perhaps this code example will bring back several nuances we learned in Chapter 3. First, the use of the function "dot" was a way to perform a dot product (weighted sum) between two vectors. However, not included in Chapter 3 is the way that numpy matrices can perform elementwise addition and multiplication.

```python
import numpy as np

a = np.array([0,1,2,1])
b = np.array([2,2,2,3])

print a*b #elementwise multiplication
print a+b #elementwise addition
print a * 0.5 # vector-scalar multiplication
print a + 0.5 # vector-scalar addition
```

One could say that numpy makes these operations very easy. When you put a "+" sign between two vectors, it does what you would expect it to. It adds the two vectors together. Other than these nice numpy operators and our new dataset, the neural network above is the same as ones we built before.

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 trained neural networks that learned how to model a single training example (input -> goal_pred pair). However, now 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!

```python
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"
```

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

Full / Batch / Stochastic Gradient Descent

Stochastic Gradient Descent - Updating weights one example at a time.

As it turns out, this idea of learning "one example at a time" is a variant on Gradient Descent called Stochastic Gradient Descent, and it is one of the handful of methods that can be used for learning an entire dataset.

How does Stochastic Gradient Descent work? As exemplified on the previous page, it simply performs a prediction and weight update for each training example separately. In other words, it takes the first streetlight, tries to predict it, calculates teh weight_delta, adn updates the weights. Then it moves onto the second streetlight, etc. It iterates through the entire dataset many times until it can find a weight configuration that works well for all of the training examples.

(Full) Gradient Descent - Updating weights one dataset at a time.

As it turns out, another method for learning an entire dataset is just called Gradient Descent (or "Average/Full Gradient Descent" if you like). Instead of updating the weights once for each training example, the network simply calculates the average weight_delta over the entire dataset, only actually changing the weights each time it computes a full average.

Batch Gradient Descent - Updating weights after "n" examples.

This will be covered in more detail later, but there is also a third configuration that sortof "splits the difference" between Stochastic Gradient Descent and Full Gradient Descent. Instead of updating the weights after just one or after the entire dataset of examples, you choose a "batch size" (typically between 8 and 256) after which the weights are updated.

We will discuss this more later in the book, but for now, simply recognize that on the previous page we created a neural network that can learn our entire "Streetlights" dataset by training on each example one at a time.

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.
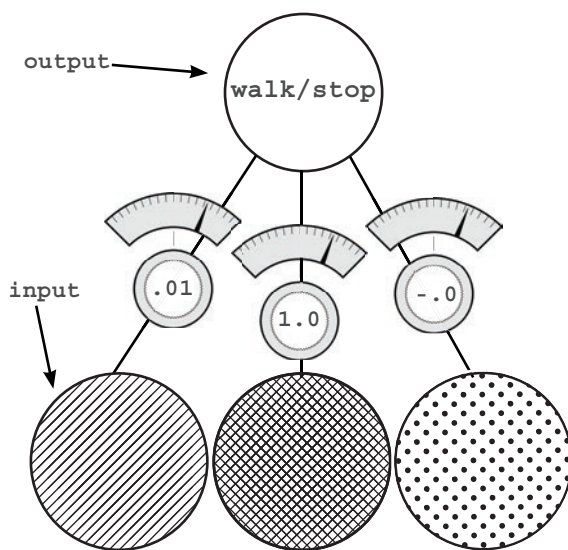
Notice that the middle weight is very near 1 while the far left and right weights are very near 0. At a high level, all the iterative, complex processes for learning we identified actually accomplished something rather simple. The network *identified correlation* between the middle input and output. 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, each neuron 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					Weight Pressure			
1	0	1		0		-	0	-	0
0	1	1	→	1		0	+	+	→ 1
0	0	1		0		0	0	-	0
1	1	1	→	1		+	+	+	→ 1
0	1	1		1		0	+	+	1
1	0	1	→	0		-	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 training 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 changed at all. 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				Weight Pressure		
1	0	1	→ 0	-	0	-	→ 0
0	1	1	→ 1	0	+	+	→ 1
0	0	1	→ 0	0	0	-	→ 0
1	1	1	→ 1	+	+	+	→ 1
0	1	1	→ 1	0	+	+	→ 1
1	0	1	→ 0	-	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 sum 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 sum of our inputs will find perfect correlation between our input and our output, by weighting de-correlated inputs to 0.

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 (i.e. those weights would fail in the real world). 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 could 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                    Weight Pressure

   1    0    1         0          -       0    -          0

   0    1    1    ⟶    1          0       +    +     ⟶    1

   0    0    1         0          0       0    -          0

   1    1    1    ⟶    1          +       +    +     ⟶    1

   0    1    1    ⟶    1          0       +    +     ⟶    1

   1    0    1         0          -       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.

As a preview, *regularization* 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?

Training Data				
1	0	1	$\rightarrow$	1
0	1	1		1
0	0	1		0
1	1	1	$\rightarrow$	0

Weight Pressure				
+	0	+		1
0	+	+	$\rightarrow$	1
0	0	−		0
−	−	−	$\rightarrow$	0

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 and 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 "know" about data. It just searches for correlation between the input and output layers.
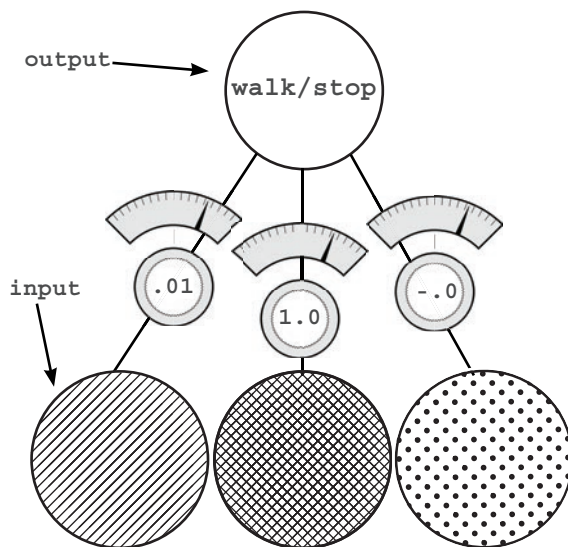
     Unfortunately, we just encountered a new streetlights dataset where there isn't *any correlation* between our input and output. The solution is simple. Let's use *two* of these networks. The first one will create an *intermediate dataset* that has *limited* correlation with our output. The second will 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.
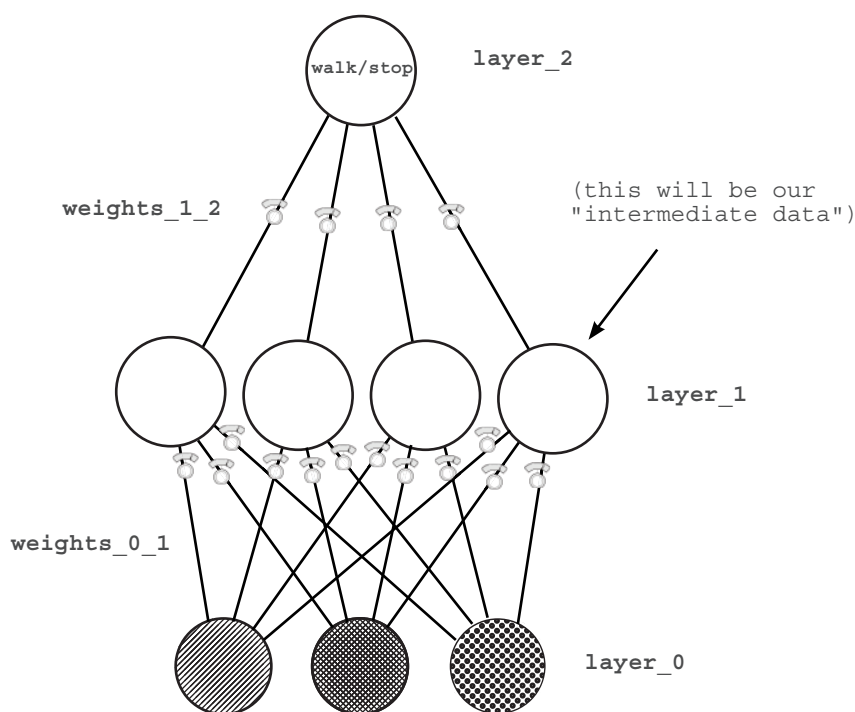
Things to notice: This network is still just a function! It still just has a bunch of weights that are collected together in a particular way. Furthermore, **Gradient Descent** still works because we can calculate how much each weight contributes to the error and adjust it to reduce the error to 0. That's exactly what we're going to do.

Stacking Neural Networks - A Review

In Chapter 3, we briefly mentioned stacked neural networks. Let's review.

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.
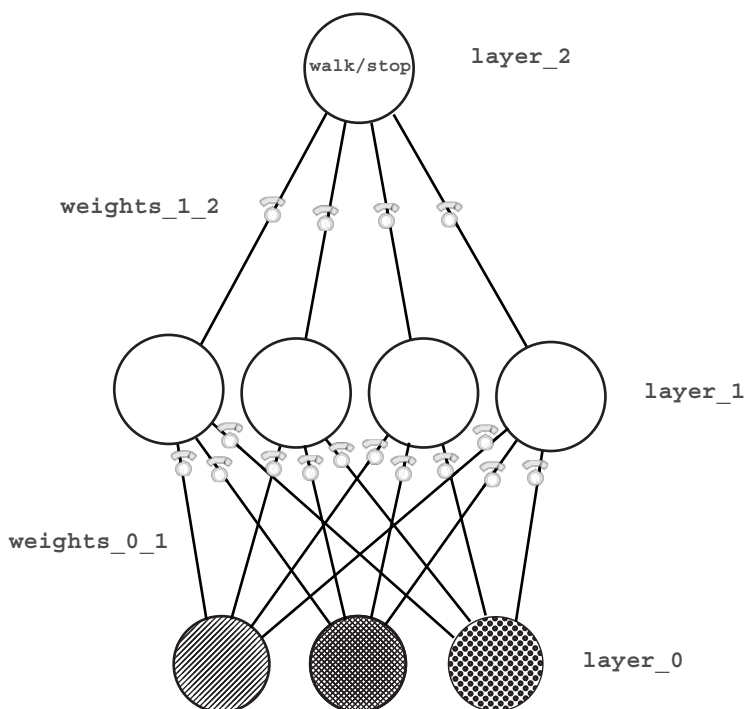
So, as we start to think about how this neural network learns, we actually already know a great deal. If we ignored the lower weights and just considered their output to be our training set, then the top half of the neural network (layer_1 to layer_2) is just like the networks we trained in the last 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 is 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**.
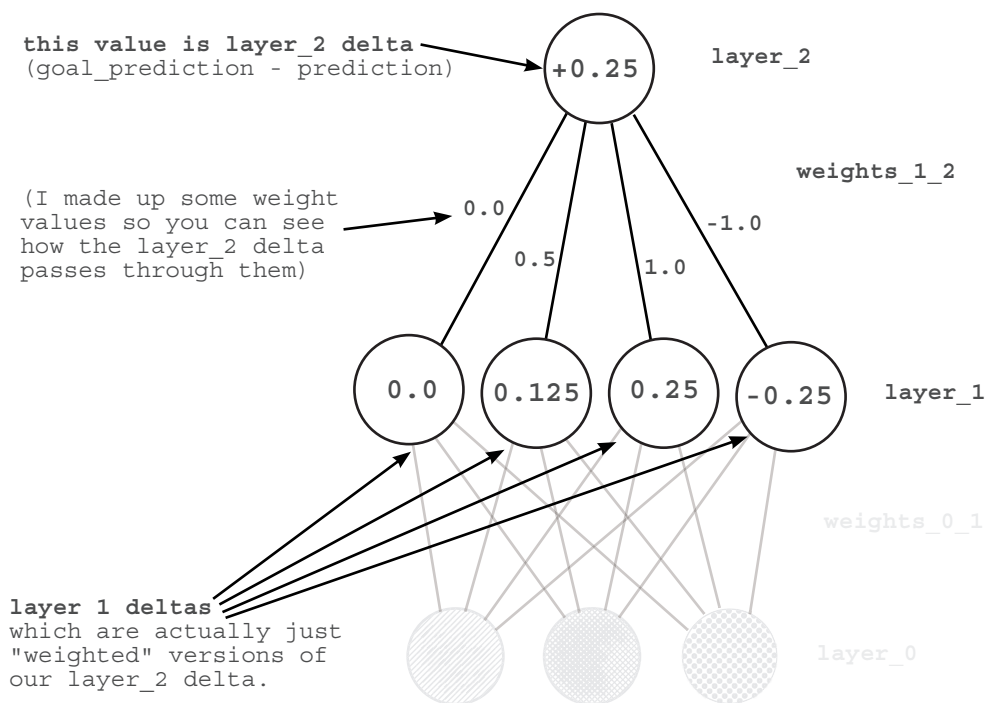
Backpropagation: Why does this work?

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 backprop-agation lets 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*).
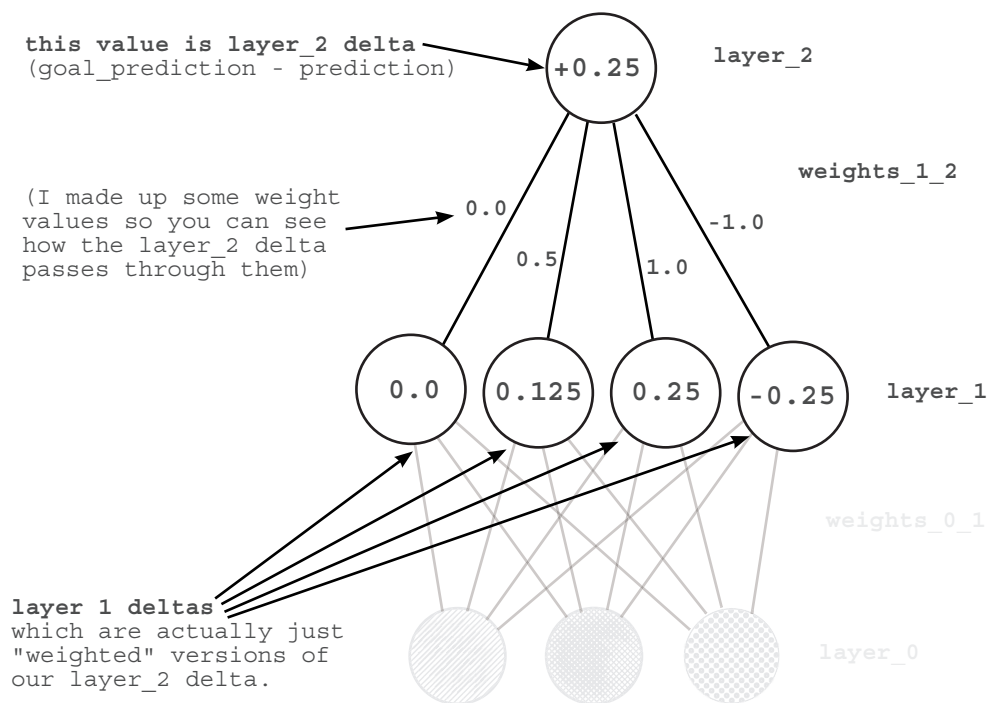
Linear vs Non-Linear

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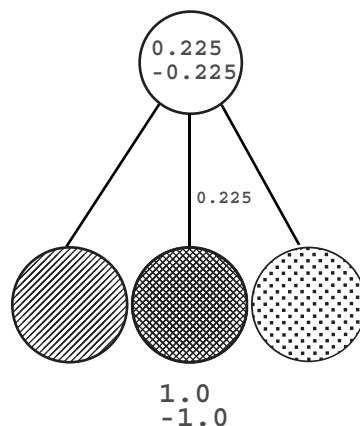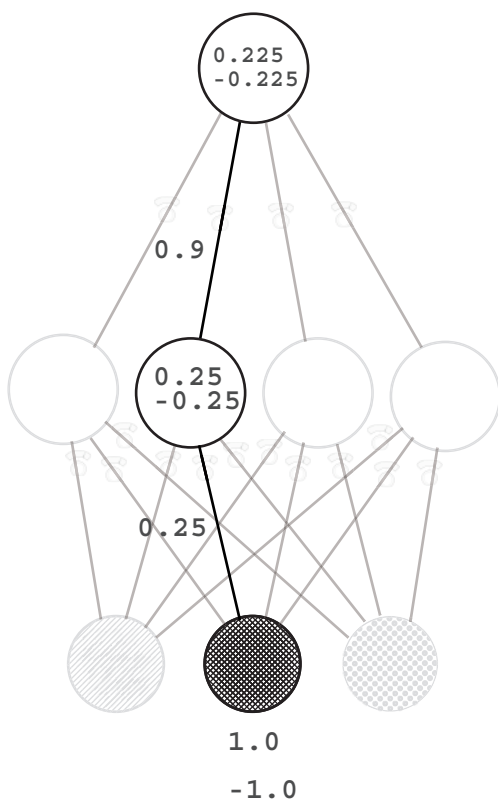
These two graphs show you *two training examples each*, one where the input is 1.0 and another where the input is -1.0. Here's the bottom line, **for any 3-layer network we create, there's a 2-layer network that has identical behavior**. It turns out that just stacking two neural nets (as we know them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

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 two nodes is 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 our 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".
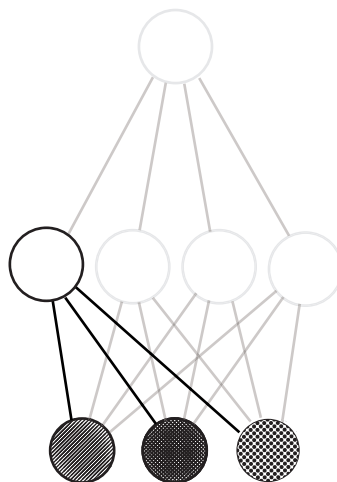
The Secret to "Sometimes Correlation"

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

This might seem too 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 in value. 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. (It's called "relu")

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 terse answer, is an intuitive explanation of why we need nonlinearities.

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 error 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 <u>correlation</u> 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 can set aside the granular and only focus on understanding 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 thousands 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. It 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 First "Deep" Neural Network

How to Make the prediction

In the code below, we initialize our weights and make a forward propagation. **New is bold.**

```
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x

alpha = 0.2
hidden_size = 4

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

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

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

layer_0 = streetlights[0]
layer_1 = relu(np.dot(layer_0,weights_0_1))
layer_2 = np.dot(layer_1,weights_1_2)
```

this function sets all negative numbers to 0

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

the output of layer_1 is sent through "relu" where negative values become 0. This is then the input for the next layer, layer_2

Take each piece and follow along with the picture on the bottom right. Input data comes into layer_0. Via the "dot" function, the signal travels up the weights from layer_0 to layer_1 (performing a weighted sum at each of the 4 layer_1 nodes). These weighted sums at layer_1 are then passed through the "relu" function, which converts all negative numbers to zero. We then perform a final weighted sum into the final node, layer_2.

layer_2

weights_1_2

layer_1

weights_0_1
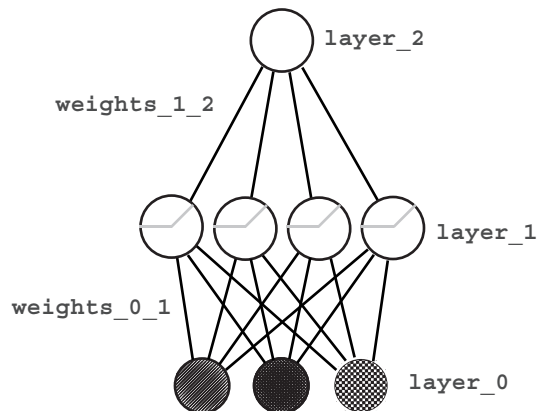
layer_0

Backpropagation in Code

How we can learn the amount that each weight contributes to the final error.

At the end of the previous chapter, I made an assertion that it would be very important to memorize the 2-layer neural network code so that you could quickly and easily recall it when I reference the more advanced concepts. <u>This is when that memorization maters!</u> We're about to look at the new learning code and it is absolutely essential that you recognize and understand the parts that were addressed in the previous chapters. If you get lost, go back to the last chapter and memorize the code and come back. It'll save your life someday.

```python
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x # returns x if x > 0
                       # return 0 otherwise

def relu2deriv(output):
    return output>0 # returns 1 for input > 0
                    # return 0 otherwise
alpha = 0.2
hidden_size = 4

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

for iteration in xrange(60):
   layer_2_error = 0
   for i in xrange(len(streetlights)):
      layer_0 = streetlights[i:i+1]
      layer_1 = relu(np.dot(layer_0,weights_0_1))
      layer_2 = np.dot(layer_1,weights_1_2)

      layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2)

      layer_2_delta = (walk_vs_stop[i:i+1] - layer_2)
      layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(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(iteration % 10 == 9):
      print "Error:" + str(layer_2_error)
```

this line computes the delta at layer_1 given the delta at layer_2 by taking the layer_2_delta and multiplying it by its connecting weights_1_2

Believe it or not, the only truly new code is in bold. Everything else is fundamentally the same as in previous pages. The "relu2deriv" function returns 1 when "output" is > 0 and it returns 0 otherwise. This is actually the *slope* of our relu function. It's the *derivative* of our relu function. It serves a very important purpose as we'll see in a moment.

Remember, the goal here is **error attribution**. It's all about figuring out how much each weight contributed to the final error. In our first (2-layer) neural network, we calculated a *delta* variable, which told us how much higher or lower we wanted the output prediction to be. Look at the code here. We compute our layer_2_delta in the same way. Nothing new here! (again, go back to the previous chapter if you've forgotten how that part works)

So, now that we have how much we want the final prediction to move up or down (delta), we need to figure out how much we want each middle (layer_1) node to move up or down. These are effectively *intermediate predictions*. Once we have the delta at layer_1, we can use all the same processes we used before for calculating a weight update (for each weight, multiply its input value by its output delta and increase the weight value by that much).
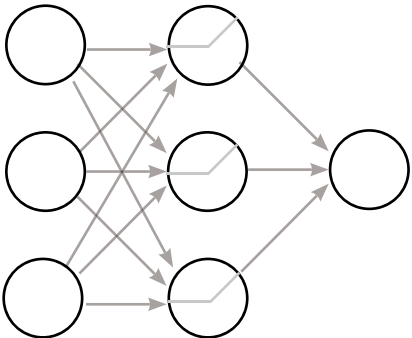
So, how do we calculate the deltas for layer_1? Well, first we do the obvious as mentioned on the previous pages, we multiply the output delta by each weight attached to it. This gives us a weighting of how much each weight contributed to that error. There's one more thing we need to factor in. If the relu set the output to a layer_1 node to be 0, then it didn't contribute to the error at all. So, when this was true, we should also set the delta of that node to be zero. Multiplying each layer_1 node by the relu2deriv function accomplishes this. relu2deriv is either a 1 or a 0 depending on whether the layer_1 value was > 0 or not.

this value is layer_2 delta
(goal_prediction - prediction)

layer_2

(I made up some weight values so you can see how the layer_2 delta passes through them)

weights_1_2

0.0   -1.0

0.5   1.0

+0.25

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
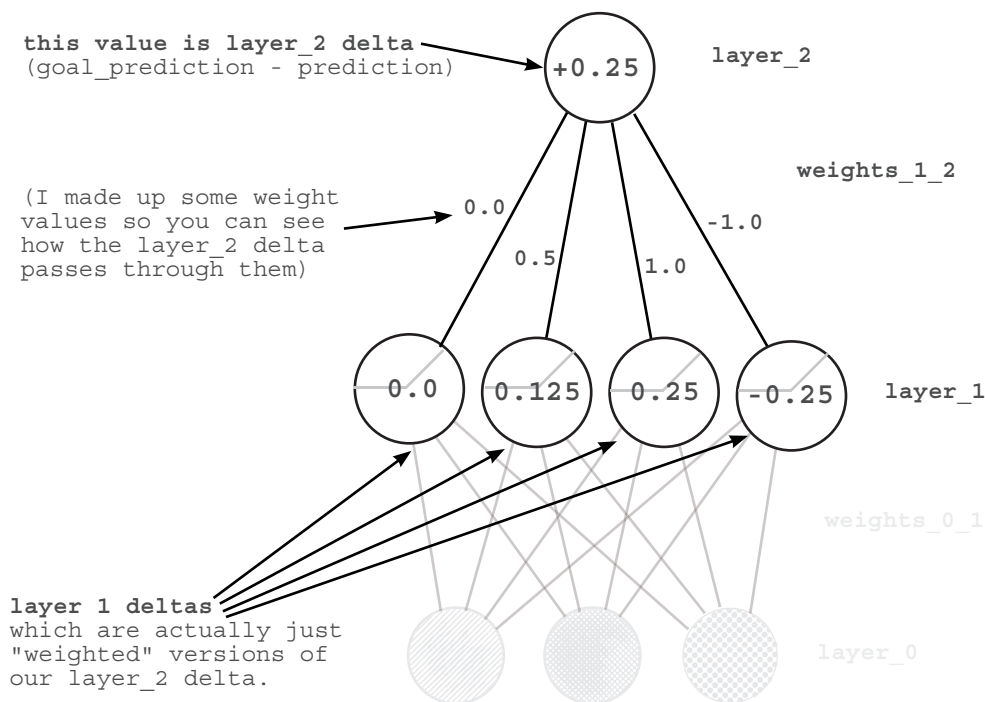
One Iteration of Backpropagation

(1) `Initialize the Network's Weights and Data`

inputs   hidden   prediction

```python
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x

def relu2deriv(output):
    return output>0

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

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

alpha = 0.2
hidden_size = 3

weights_0_1 = 2*np.random.random(\
              (3,hidden_size)) - 1
weights_1_2 = 2*np.random.random(\
              (hidden_size,1)) - 1
```
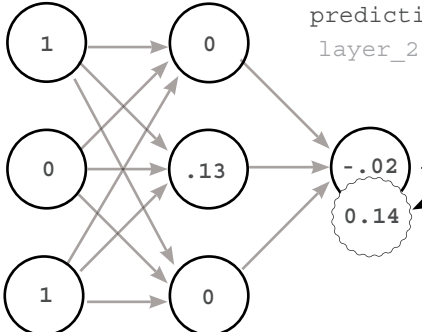
(2) `PREDICT & COMPARE: Make a Prediction, Calculate Output Error and Delta`

inputs   hiddens
layer_0   layer_1

```python
layer_0 = lights[0:1]
layer_1 = np.dot(layer_0,weights_0_1)
layer_1 = relu(layer_1)
layer_2 = np.dot(layer_1,weights_1_2)

error = (layer_2-walk_stop[0:1])**2

layer_2_delta=(layer_2-walk_stop[0:1])
```

prediction
layer_2

1   0

0   .13   -.02   1.04
                        0.14

1   0

(3) **LEARN: Backpropagate From layer_2 to layer_1**

inputs   hiddens
layer_0   layer_1

```
layer_0 = lights[0:1]
layer_1 = np.dot(layer_0,weights_0_1)
layer_1 = relu(layer_1)
layer_2 = np.dot(layer_1,weights_1_2)
```

prediction   `error = (layer_2-walk_stop[0:1])**2`
layer_2
            `layer_2_delta=(layer_2-walk_stop[0:1])`

```
layer_1_delta=layer_2_delta.dot(weights_1_2.T)
layer_1_delta *= relu2deriv(layer_1)
```
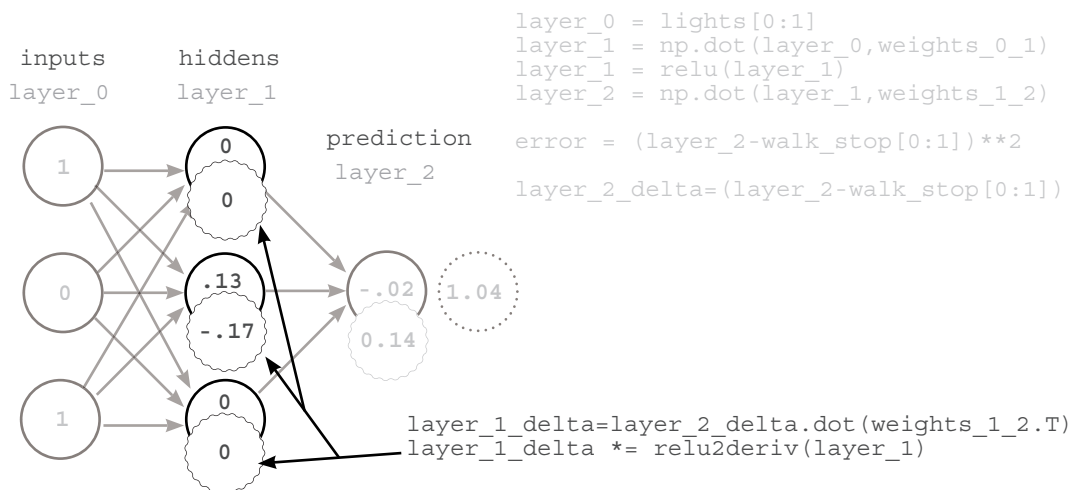
(4) **LEARN: Generate Weight Deltas and Update Weights**

inputs   hiddens   prediction
layer_0   layer_1   layer_2

```
layer_0 = lights[0:1]
layer_1 = np.dot(layer_0,weights_0_1)
layer_1 = relu(layer_1)
layer_2 = np.dot(layer_1,weights_1_2)
error = (layer_2-walk_stop[0:1])**2
layer_2_delta=(layer_2-walk_stop[0:1])
```

```
layer_1_delta=layer_2_delta.dot(weights_1_2.T)
layer_1_delta *= relu2deriv(layer_1)
```

```
weight_delta_1_2 = layer_1.T.dot(layer_2_delta)
weight_delta_0_1 = layer_0.T.dot(layer_1_delta)

weights_1_2 -= alpha * weight_delta_1_2
weights_0_1 -= alpha * weight_delta_0_1
```

As we can see, backpropagation in its entirety is about calculating deltas for intermediate layers so that we can perform Gradient Descent. In order to do so, we simply take the weighted average delta on layer_2 for layer_1 (weighted by the weights inbetween them). We then turn off (set to 0) nodes that weren't participating in the forward prediction, since they could not have contributed to the error.

Putting it all together

Here's the self sufficient program you should be able to run (runtime output below)

```python
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x # returns x if x > 0
                       # return 0 otherwise

def relu2deriv(output):
    return output>0 # returns 1 for input > 0
                    # return 0 otherwise

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

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

alpha = 0.2
hidden_size = 4

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

for iteration in xrange(60):
   layer_2_error = 0
   for i in xrange(len(streetlights)):
       layer_0 = streetlights[i:i+1]
       layer_1 = relu(np.dot(layer_0,weights_0_1))
       layer_2 = np.dot(layer_1,weights_1_2)

       layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2)

       layer_2_delta = (layer_2 - walk_vs_stop[i:i+1])
       layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(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(iteration % 10 == 9):
       print "Error:" + str(layer_2_error)
```

```
Error:0.634231159844
Error:0.358384076763
Error:0.0830183113303
Error:0.0064670549571
Error:0.000329266900075
Error:1.50556226651e-05
```

Why do deep networks matter?

What's the point of creating "intermediate datasets" that have correlation?

Consider the cat picture below. Consider further that we had a dataset of images "with cats" and "without cats" (and we labeled them as such). If we wanted to train a neural network to take our pixel values and predict whether or not there is a cat in the picture, our 2-layer network might have a problem. Just like in our last streetlight dataset, no individual pixel correlates with whether or not there is a cat in the picture. Only different *configurations of pixels* correlate with whether or not there is a cat.

This is the essence of Deep Learning. Deep Learning is all about creating intermediate layers (datasets) wherein each node in an intermediate layer represents the *presence* or *absence* of a different *configuration of inputs*. In this way for our cat images dataset, no individual pixel has to correlate with whether or not there is a cat in the photo. Intead, our middle layer would attempt to identify different configurations of pixels that may or may not correlate with a cat (such as an ear, or cat eyes, or cat hair). The presence of many "cat like" configurations would then give the final layer the information (correlation) it needs to correctly predict the presence of absence of a cat!

Believe it or not, we can take our 3-layer network and continue to *stack more and more layers*. Some neural networks even have *hundreds* of layers, each neuron playing its part in detecting different configurations of input data. The rest of this book will be dedicated to studying different phenomena within these layers in an effort to explore the full power of deep neural networks.

To that end, I must issue the same challenge as I did in the previous chapter. Memroze the code on the previous page. You will need to be *very familiar* with each of the operations in the code in order for the following chapters to be readable. Do not progress past this page until you can build a 3 layer neural network from memory!!!

How to Picture Neural Networks
In Your Head and on Paper

7

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*

It's Time to Simplify

It's impractical to think about everything all the time. Mental tools can help.

At the end of the previous chapter, we finished with a code example that was really quite impressive. Just the neural network itself contained 35 lines of incredibly dense code. Reading through it, it's clear that there's a lot going on, and contained in that code is over 100 pages of concepts which when combined together can predict whether it's safe to cross the street.

I hope that you're continuing to rebuild these examples from memory in each chapter. As these examples get larger, this exercise becomes less about remembering specific letters of code and more about remembering the concepts and then re-building the code based on those concepts. In this chapter, this construction of efficient concepts in your mind is exactly what I want to set aside time to talk about. Even though it's not some architecture or experiment, it's perhaps the most important value I can give you. In this case, I want to show you how I summarize all of these little lessons in an efficient way in my mind so that I can do things like build new architectures, debug experiments, and leverage an architecture on new problems and new datasets.

Let's start by reviewing the concepts we've learened so far.

In the beginning of this book, we started with very small lessons and then built layers of abstraction on top of them. We began by talking about the ideas behind Machine Learning in general. Then, we progressed to how individual linear neurons learned, followed by horizontal groups of neurons (layers) and then vertical groups (stacks of layers). Along the way, we discussed how learning is actually just reducing error down to 0, and we then leveraged Calculus to learn how to change each weight in our network to help move our error in the direction of 0. In the last chapters, we then discussed how neural networks actually search for (and sometimes even create) correlation between the input and output datasets. This last idea allowed us to overlook the previous lessons on how individual neurons behaved because it concisely summarizes the previous lessons. The sum total of the neurons, gradients, stacks of layers, etc. all lead up to a single idea: neural networks find and create correlation. Holding onto this idea of correlation instead of the previous smaller ideas is important to learning Deep Learning. Otherwise, it would be easy to become overwhelmed with the complexity of neural networks. In fact, let's create a name for his idea. Let's call this our Correlation Summarization.

Correlation Summarization

This is the key to sanely moving forward to more advanced neural networks.

Correlation Summarization

Neural networks seek to find direct and indirect correlation between an input layer and an output layer, which are determined by the input and output datasets respectively.

At the 10,000 ft. level, this is what all neural networks do. So, given that a neural network is really just a series of matrices connected by layers, let's zoom in slightly and consider what any particular weight matrix is doing.

Local Correlation Summarization

Any given set of weights optimizes to learn how to correlate its input layer with what the output layer says it should be.

When we only have two layers (input and output), this means that our weight matrix knows what the output layer says it should be based on the output dataset. It looks for correlation between the input and output datasets because they are captured in the input and output layers. However, this becomes more nuanced when we have multiple layers, remember?

Global Correlation Summarization

What an earlier layer says it should be can be determined by taking what a later layer says it should be and multiplying it by the weights inbetween them. In this way, later layers can tell earlier layers what kind of signal they need to ultimately find correlation with the output. We call this cross-communication, backpropagation.

 Thus, when global correlation teaches each layer what it should be, then local correlation can optimize weights locally. In other words, given that some neuron in the final layer says "I need to be a little higher", it then proceeds to tell all the neurons in the layer immediately preceeding it, "Hey previous layer, send me higher signal!". They then tell the neurons preceeding them "Hey! Send us higher signal!". It's like a giant game of telephone! At the end of the game, every layer knows which of its neurons need to be higher and lower, and the Local Correlation Summarization takes over, updating the weights accordingly.

Our Previously Overcomplicated Visualization

While we simplify our mental picture, let's simplify our visualization as well.

At this point, I would expect that the visualization of neural networks in your head is something like the picture in the bottom right (because that's the one we used). We have our input dataset in layer_0, connected by a weight matrix (bunch of lines) to layer_1 and so on. This was a very useful tool to learn the basics of how collections of weights and layers come together to learn a function.

However, moving forward, this picture simply has too much detail. Given our correlation summarization, we already know that we no longer need to worry about how individual weights are updated. Later layers already know how to communicate to earlier layers and tell them "hey, I need higher signal" or "hey, I need lower signal". Truth be told, we don't really care about the actual weight values anymore, only that they're behaving how they should, properly capturing correlation in a way that generalizes.

So, to refect the change in our attention, we need to update our visualization on paper. We're also going to do a few more things which will make sense later. As you know, our neural network is really just a series of weight matrices. When we're using the network, we also end up creating vectors corresponding to each layer. In our picture, the weight matrices are the lines going from node to node, and the vectors are the strips of nodes themselves. For example, weights_1_2 is a matrix, weights_0_1 is a matrix, and layer_1 is a vector. In later chapters, we're going to be arranging vectors and matrices in increasingly creative ways, so instead of all this detail showing each node connected by each weight (which gets hard to read if we have...say... 500 nodes in layer_1), let's instead just think in general terms. Let's think of them as just vectors and matrices of arbitrary size.

layer_2

weights_1_2

("relu" nodes on this layer)

layer_1

weights_0_1

layer_0
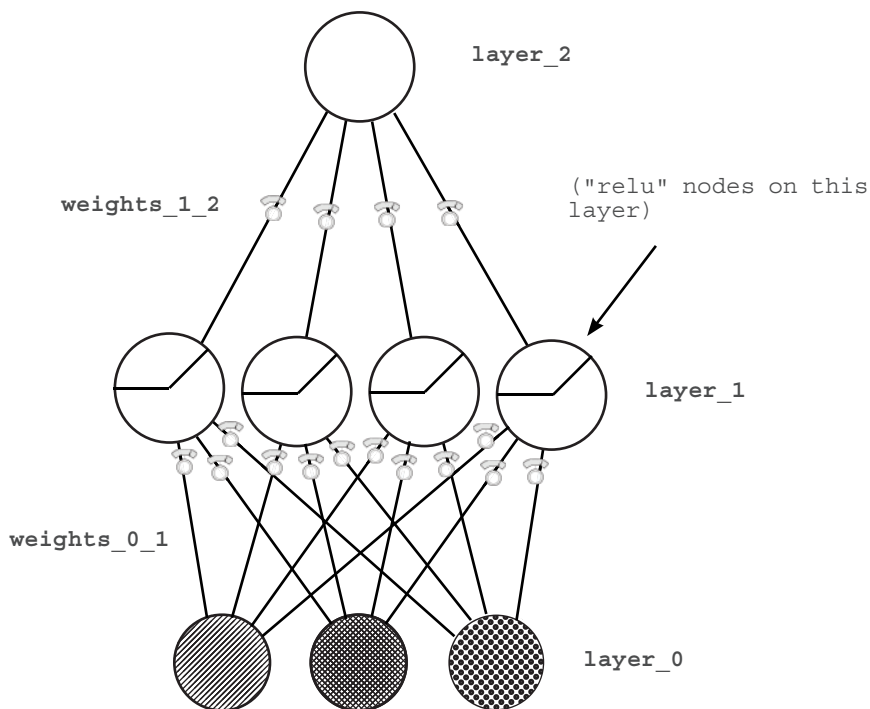
Our Simplified Visualization

Neural networks are like legos, and each block is a vector or matrix.

Moving forward, we're going to be building new neural network architectures in the same way that people build new structures with the same lego pieces. The great thing about our correlation summarization is that all the bits and pieces that lead to it (backpropagation, gradient descent, alpha, dropout, mini-batching, etc.) don't really depend on a particular configuration of our legos! No matter how we piece together our series of matrices, gluing them together with layers, our neural network will try to learn the pattern in our data by modifying the weights between wherever we put the input layer and the output layer. To reflect this, we're going to build all our neural networks with these lego pieces you see on the right. The strip is a vector. The box is a matrix. The circles are individual weights. Note that the box can be viewed as a "vector of vectors" horizontally or vertically!

numbers   vector

matrix

(1x1) layer_2

(4x1) weights_1_2

(1x4) layer_1

(3x4) weights_0_1

(1x3) layer_0
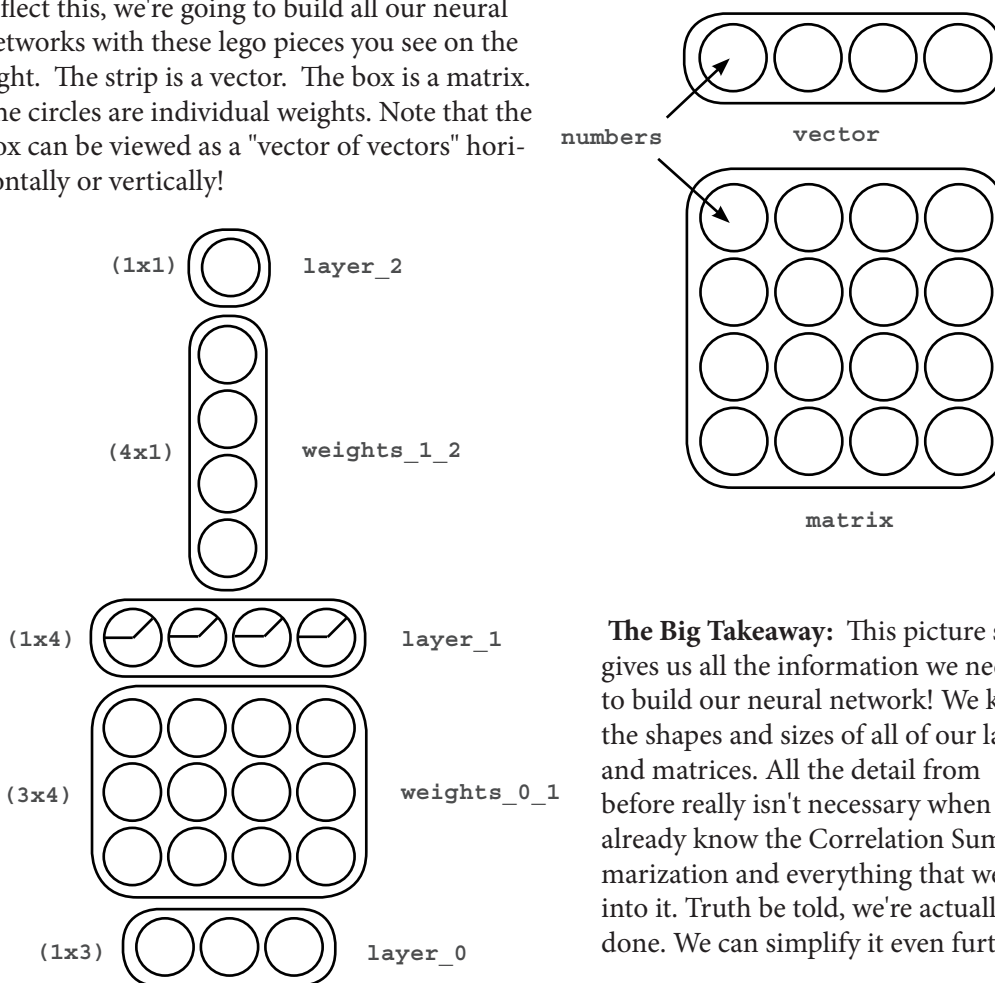
The Big Takeaway: This picture still gives us all the information we need to build our neural network! We know the shapes and sizes of all of our layers and matrices. All the detail from before really isn't necessary when you already know the Correlation Summarization and everything that went into it. Truth be told, we're actually not done. We can simplify it even further!

Simplifying Even Further

The dimensionality of our matrices are determined by the layers!

On the previous page, you might have noticed a pattern. Each matrix's dimensionality (number of rows and columns) has a direct relationship to the dimensionality of the layers before and after them! Thus, we can actually simplify our visualization even further!

Consider the visualization on the right. We still have all the information we need to build the neural network. We can infer that weight_0_1 is a (3x4) matrix because the previous layer (layer_0) has 3 diensions and the next layer (layer_1) has 4 dimensions. Thus, in order for the matrix to be big enough to have a single weight connecting each node in layer_0 to each node in layer_1, it must be a (3x4) matrix!

This allows us to really start thinking about our neural networks using the correlation summarization. All this neural network is going to do is adjust the weights to find correlation between layer_0 and layer_2. It's going to do this using all the methods we've mentioned so far in this book. However, we will find that the different configurations of weights and layers between our input and output layers have a strong impact on whether or not the network is successful in finding correlation (and/or how fast it finds correlation).

The particular configuration of layers and weights in a neural network is called its **architecture**, and we will spend the majority of the rest of this book discussing the pros and cons of various architectures. As the correlation summarization reminds us, the neural network adjusts weights to find correlation between the input and output layers, sometimes even inventing correlation in the hidden layers. We will find that different architectures *channel signal to make correlation easier to discover*.
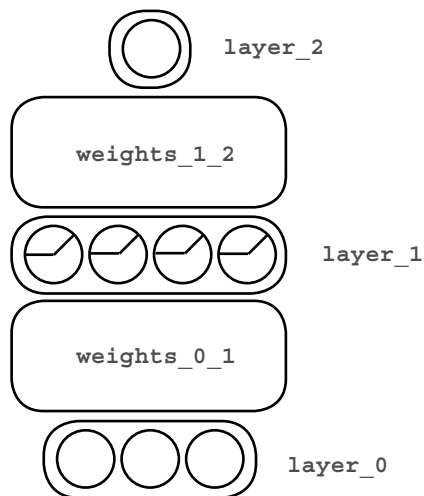
> Good neural architectures channel signal so that correlation is easy to discover. Great architectures also filter noise to help prevent overfitting.

Much of research into neural networks is about finding new architectures that can find correlation faster and generalize better to unseen data. We will spend the vast majority of the rest of this book discussing new architectures.

Let's See This Network Predict

Let's picture data from our streetlight example flowing through the system.

1 A single datapoint from our "streetlight" dataset is selected. layer_0 is set to the correct values. (pictured right)

2 Four different weighted sums of layer_0 are performed. The four weighted sums are performed by weights_0_1. As a reminder, this process is called *vector-matrix multiplication.* These four values are deposited into the four positions of layer_1 and passed through the "relu" function (setting negative values to 0). To be clear, the 3rd value from the left in layer_1 would have been negative, but the "relu" function sets it to 0. (pictured left)

3 The final step (pictured bottom right) performs a weighted average of layer_1, again using the *vector-matrix multiplication* process. This yields the number 0.9, which is the network's final prediction.
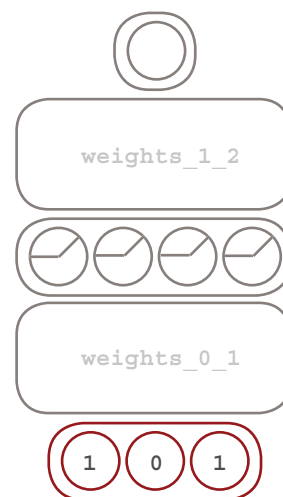
Review: Vector-Matrix Multiplication

Vector-Matrix multiplication performs multiple *weighted sums* of a vector. The matrix must have the same number of rows as the vector has values, so that each column in the matrix performs a unique weighted sum. Thus, if the matrix has 4 columns, 4 weighted sums will be generated. The weightings of each sum are performed depending on the values of the matrix.

Visualizing Using Letters Instead of Pictures

All these pictures and detailed explanations are actually a simple piece of algebra.

Just like we defined simpler pictures for our matrix and vector, we can actually perform the same visualization in the form of letters. Let me show you.

matrix

W_0

vector

l_0
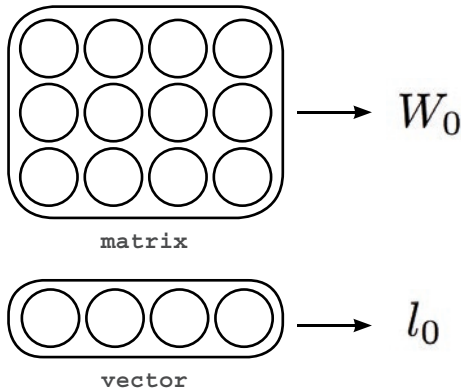
So, how do we visualize a matrix using math? Well, we simply pick a capital letter! I try to pick one that's easy to remember such as "W" for "weights". The little 0 down there? That just means it's probably one of several Ws. In our case, our network has 2. Perhaps surprisingly, I literally could have picked any capital letter, and the little 0 at the bottom is extra. It just lets me call all my weight matrices W so I can keep them apart. It's your visualization! Make it easy to remember!

How do we visualize a vector using math? Well, we pick a lowercase letter! Why did I choose the letter "L"? Well, since I have a bunch of vectors that are "layers", I thought "L" would be easy to remember! Why did I choose to call it "L"-zero? Well, Since I have multiple "layers", it seems nice to make all of them "L"s and just number them instead of having to think of new letters for every layer. There's no wrong answer here! How bout that? So, if that's how we visualize matrices and vectors, in math, what do all the pieces in our network look like? Well, to the right you can see a nice selection of variables pointing to their respective sections of the neural network. However, simply defining them doesn't show how they relate. Let's combine our variables together via vector-matrix multiplication!

l_2

weights_1_2

W_1

l_1

weights_0_1

W_0

l_0
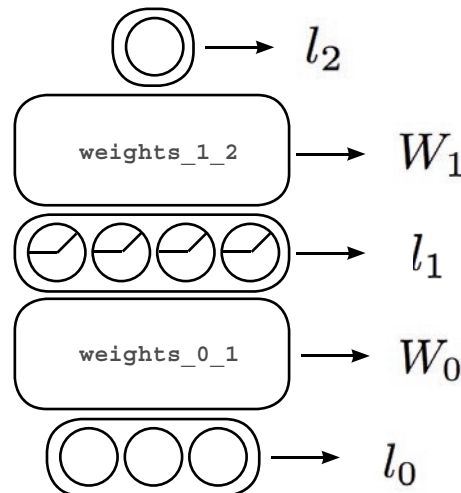
Linking Our Variables

Our letters can be combined together to indicate functions and operations.

Vector-matrix multiplication is very simple. If you want to visualize that two letters are being multiplied by each other, you literally just put them next to each other. For example:

algebra	**translation**
$l_0 W_0$	"take the layer 0 vector and perform vector-matrix multiplication with the weight matrix 0"
$l_1 W_1$	"take the layer 1 vector and perform vector-matrix multiplication with the weight matrix 1"

We can even throw in arbitrary functions like "relu" using notation that looks almost exactly like the Python code! This really is crazy intuitive stuff!

$$l_1 = relu(l_0 W_0)$$

"to create the layer 1 vector, take the layer 0 vector and perform vector-matrix multiplication with the weight matrix 0, then perform the "relu" function on the output (setting all negative numbers to 0)."

$$l_2 = l_1 W_1$$

"to create the layer 2 vector, take the layer 1 vector and perform vector-matrix multiplication with the weight matrix 1"

If you notice, the layer 2 algebra actually contains layer 1 as an input variable. This means we can actually represent our *entire neural network* in one expression by just chaining them together. Thus, all the logic in our forward propagation step can be contained in this one formula in the bottom left corner of this page. Note: baked into this formula is the assumption that our vectors and matrices have the right dimensions.

$$l_2 = relu(l_0 W_0) W_1$$

Everything Side-by-Side

Let's see our visualization, algebra formula, and python code in one place.

I don't think too much dialogue is necessary on this page. Just take a minute and look at each piece of forward propagation through these 4 different ways of seeing it. It is my hope that you would truly grok forward propagation. Understand the architecture by seeing it from different perspectives, all in one place.

```
layer_2 = relu(layer_0.dot(weights_0_1)).dot(weights_1_2)
```

$$l_2 = relu(l_0 W_0) W_1$$
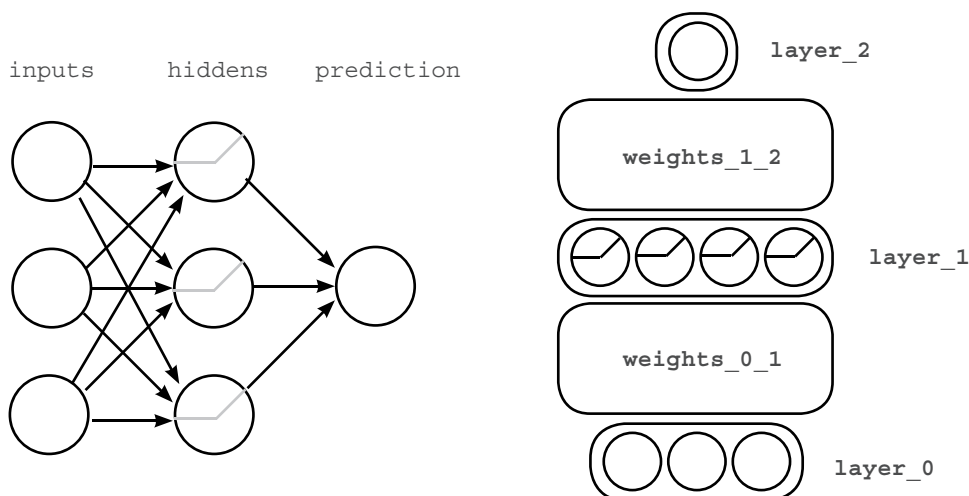
The Importance of Visualization Tools

We're going to be studying new architectures.

In the following chapters, we're going to be taking these vectors and matrices and combining them in some very creative ways. My ability to describe each architecture for you is entirely dependent on our ability to have a mutually agreed upon language for describing them. Thus, please don't move beyond this chapter until you can clearly see how forward propagation manipulates these vectors and matrices, and how these various forms of describing them are articulated.

> Good neural architectures channel signal so that correlation is easy to discover. Great architectures also filter noise to help prevent overfitting.

As mentioned previously, a neural architecture controls how signal flows through a network. How we create these architectures will affect the ways in which the network can detect correlation. We will find that we want to create architectures that maximise the network's ability to focus on the areas where meaningful correlation exist, and minimize the network's ability to focus on the areas that simply contain noise.

However, different datasets and domains have different characteristics. For example, image data has different kinds of signal and noise than text data. Thus, we will find that even though neural networks can be used in many situations, different architectures will be better suited to different problems because of their ability to locate certain types of correlations. So, for the next few chapters, we're going to explore how you can modify neural networks to specifically find the correlation you're looking for. See you there!

Learning Signal and Ignoring Noise
Introduction to Regularization & Batching

8

IN THIS CHAPTER ·······························

Overfitting

Dropout

Batch Gradient Descent

> ❝ This is a sample quote that I'm trying to make, but Andrew... would you like to perhaps put something else here? ❞
>
> — *SOME AUTHOR*

3 Layer Network on MNIST

Let's return to our MNIST dataset and attempt to classify it with our new network

In last several chapters, we have learned that neural networks model correlation. In fact, the hidden layers (the middle one in our 3 layer network) can even create "intermediate" correlation to help solve for a task (seemingly out of mid air). How do we know that our network is creating good correlation?

Back when we learned about Stochastic Gradient Descent with Multiple Inputs, we ran an experiment where we froze one weight and then asked our network to continue training. As it was training, we watch the "dots find the bottom of the bowls" as it were. We were watching the weights become adjusted to minimize the error.

When we froze the weight, however, we were surprised to see that the frozen weight still found the bottom of the bowl! For some reason, the bowl moved so that the frozen weight value becamse optimal. Furthemore, if we unfroze the weight after training to do some more training, it wouldn't learn! Why? Well, the error had already fallen to 0! As far as the network was concerned, there was nothing more to learn!

This begs the question, what if the input to the frozen weight was actually important to predicting baseball victory in the real world? What if the network had figured out a way to accurately predict the games in the training dataset (because that's what networks do, they minimize the error), but somehow forgot to include a valuable input?

This phenomenon is, unfortunately, extremely common in neural networks. In fact, on might say it's "The Arch Nemesis" of neural networks, **Overfitting**, and unfortunately, the more powerful your neural network's expressive power (more layers / weights), the more prone the network is to overfit. So, there's an everlasting battle going on in research where people continually find tasks that need more powerful layers but find themselves having to do lots of problem solving to make sure the network doesn't "overfit".

In this chapter, we're going to study the basics of Regularization, which are key to combatting overfitting in neural networks. In order to do this, we're going first start with our most powerful neural network (3 layer network with relu hidden layer) on our most challenging task (MNIST digit classification).

So, to start, go ahead and train the network on the following page. You should see the same results as those listed below. Alas! Our network learned to perfectly predict the training data! Should we celebrate?

```
import numpy as np

np.random.seed(1)

def relu(x):
    return (x >= 0) * x # returns x if x > 0
                        # return 0 otherwise

def relu2deriv(output):
    return output >= 0 # returns 1 for input > 0
                       # return 0 otherwise

alpha = 0.005
iterations = 300
hidden_size = 40
pixels_per_image = 784
num_labels = 10

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in xrange(iterations):

    error = 0.0
    correct_cnt = 0

    for i in xrange(len(images)):

        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[i:i+1] - layer_2) ** 2)

        correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(labels[i:i+1]))

        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
                                    * relu2deriv(layer_1)

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

    sys.stdout.write("\r"+ \
                    " I:"+str(j)+ \
                    " Error:" + str(error/float(len(images)))[0:5] +\
                    " Correct:" + str(correct_cnt/float(len(images))))
```

```
 ....
 I:299 Error:0.101 Correct:1.0
```

Well... that was easy!

Our neural network perfectly learned to predict all 1000 images!

So, in some ways, this is a real victory. Our neural network was able to take a dataset of 1000 images and learn to correlate each input image with the correct label. How did it do this? Well, it simply iterated through each image, made a prediction, and then updated each weight ever so slightly so that the prediction was better next time. Doing this long enough on all the images eventually reached a state where the network could correctly predict on all of the images!

Perhaps a non-obvious question: how well will the neural network do on an image that it hasn't seen before? In other words, how well will it do on an image that wasn't part of the 1000 images it was trained on? Well, our MNIST dataset has many more images than just the 1000 we trained on; let's try it out! In the same notebook from before "Chapter 6 Neural Networks in the Real World" there are two variables called images_test and labels_test. If you execute the following code, it will run the neural network on these images and evaluate how well the network classifies them.

```
error = 0.0
correct_cnt = 0

for i in xrange(len(test_images)):

    layer_0 = test_images[i:i+1]
    layer_1 = relu(np.dot(layer_0,weights_0_1))
    layer_2 = np.dot(layer_1,weights_1_2)

    error += np.sum((test_labels[i:i+1] - layer_2) ** 2)

    correct_cnt += int(np.argmax(layer_2) == \
                                  np.argmax(test_labels[i:i+1]))
sys.stdout.write("\r"+ \
                " Error:" + str(error/float(len(test_images)))[0:5] +\
                " Correct:" + str(correct_cnt/float(len(test_images))))
```

```
   Error:0.803 Correct:0.601
```

The network did horribly! It only predicted with an accuracy of 60.1%. Why does it do so terribly on these new testing images when it learned to predict with 100% accuracy on the training data? How strange! We call this 60.1% number the *test accuracy.* It's the accuracy of the neural network on data that the network was NOT trained on. This number is important because it simulates how well our neural network will perform if we tried to use it in the real world (which only gives us images we haven't seen before). This is the score that matters!

Memorization vs Generalization

"Memorizing" 1000 images is easier than "Generalizing" to all images.

Let's consider again how the neural network learns. It adjusts each weight in each matrix so that the network is better able to take *specific inputs* and make a *specific prediction*. Perhaps a better question might be, "If we train it on 1000 images which it learns to predict perfectly, why does it work on other images at all?" As you might expect, when our fully trained neural network is applied to a new image, it is only guaranteed to work well if the new image is *nearly identical to an image from the training data*. Why? Well, the neural network only learned to transform input data to output data for *very specific input configurations*. If you give it something that doesn't look *familiar*, then it will predict randomly!

Well this makes neural networks kindof pointless! What's the point in a neural network only working on the data you trained it on! You already know the correct classifications for those datapoints! Neural networks are only useful if they work on data we don't already know the answer to! As it turns out, there's a way to combat this. For a clue, below I've printed out BOTH the training *and testing accuracy* of the neural network *as it was training* (every 10 iterations). Notice anything interesting? You should see a clue to better networks!

```
I:0 Test-Err:0.650 Test-Acc:0.595 Train-Err:0.743 Train-Acc:0.511
I:10 Test-Err:0.471 Test-Acc:0.746 Train-Err:0.293 Train-Acc:0.917
I:20 Test-Err:0.478 Test-Acc:0.753 Train-Err:0.222 Train-Acc:0.952
I:30 Test-Err:0.501 Test-Acc:0.74 Train-Err:0.189 Train-Acc:0.974
I:40 Test-Err:0.527 Test-Acc:0.728 Train-Err:0.170 Train-Acc:0.986
I:50 Test-Err:0.552 Test-Acc:0.708 Train-Err:0.158 Train-Acc:0.991
I:60 Test-Err:0.574 Test-Acc:0.692 Train-Err:0.150 Train-Acc:0.992
I:70 Test-Err:0.595 Test-Acc:0.676 Train-Err:0.144 Train-Acc:0.993
I:80 Test-Err:0.614 Test-Acc:0.661 Train-Err:0.138 Train-Acc:0.995
I:90 Test-Err:0.633 Test-Acc:0.651 Train-Err:0.134 Train-Acc:0.996
I:100 Test-Err:0.650 Test-Acc:0.645 Train-Err:0.129 Train-Acc:0.996
I:110 Test-Err:0.667 Test-Acc:0.635 Train-Err:0.126 Train-Acc:0.997
I:120 Test-Err:0.682 Test-Acc:0.633 Train-Err:0.123 Train-Acc:0.998
I:130 Test-Err:0.697 Test-Acc:0.631 Train-Err:0.121 Train-Acc:0.998
I:140 Test-Err:0.711 Test-Acc:0.627 Train-Err:0.119 Train-Acc:0.999
I:150 Test-Err:0.722 Test-Acc:0.627 Train-Err:0.117 Train-Acc:0.999
I:160 Test-Err:0.733 Test-Acc:0.625 Train-Err:0.116 Train-Acc:0.999
I:170 Test-Err:0.742 Test-Acc:0.62 Train-Err:0.114 Train-Acc:0.999
I:180 Test-Err:0.750 Test-Acc:0.616 Train-Err:0.112 Train-Acc:0.999
I:190 Test-Err:0.758 Test-Acc:0.614 Train-Err:0.111 Train-Acc:0.999
I:200 Test-Err:0.765 Test-Acc:0.612 Train-Err:0.110 Train-Acc:0.999
I:210 Test-Err:0.771 Test-Acc:0.611 Train-Err:0.108 Train-Acc:0.999
I:220 Test-Err:0.776 Test-Acc:0.612 Train-Err:0.107 Train-Acc:0.999
I:230 Test-Err:0.779 Test-Acc:0.611 Train-Err:0.106 Train-Acc:0.999
I:240 Test-Err:0.783 Test-Acc:0.61 Train-Err:0.105 Train-Acc:0.999
I:250 Test-Err:0.786 Test-Acc:0.605 Train-Err:0.104 Train-Acc:0.999
I:260 Test-Err:0.790 Test-Acc:0.606 Train-Err:0.103 Train-Acc:0.999
I:270 Test-Err:0.793 Test-Acc:0.603 Train-Err:0.103 Train-Acc:0.999
I:280 Test-Err:0.796 Test-Acc:0.6 Train-Err:0.102 Train-Acc:0.999
I:290 Test-Err:0.800 Test-Acc:0.602 Train-Err:0.101 Train-Acc:1.0
```

Overfitting in Neural Networks

Neural networks can get worse if we train them too much?

For some reason, our *test* accuracy went up for the first 20 iterations, and then slowly decreased as the network trained more and more (during which time the *training* accuracy was still improving). This is very common in neural networks. Let me explain the phenomenon via analogy.

Imagine that you are creating a *mold* for a common dinner fork, but instead of using it to create other forks you want to use it to *identify* if a particular utensil is in fact, a fork. If an object fits in the mold, you would conclude that the object is a fork, and if it does not, then you would conclude that it is *not* a fork.

Let's say you set out to make this mold, and you start with a wet piece of clay and a big bucket of 3-pronged forks, spoons, and knives. You then press each of the forks into the same place in the mold to create an outline, which sortof looks like a mushy fork. You repeatedly place all the forks in the clay over and over, hundreds of times. When you let the clay dry, you then find that none of the spoons or knives fit into this mold, but all of the forks fit into this mold. Awesome! You did it! You correctly made a mold that can only fit the shape of a fork!

However, what happens if someone hands you a 4-pronged fork? You look in your mold and notice that there is a specific outlines for three, thin prongs in your clay. Your 4-pronged fork doesn't fit! Why not? It's still a fork!

This is because the clay wasn't molded on any 4-pronged forks. It was only molded on the on 3-pronged variety. In this way, the clay has **overfit** to only recognize the types of forks that it was "trained" to shape.

This is exactly the same phenomenon that we just witnessed in our neural newtork. It's actually an even closer parallel than you might think. In truth, one way to view the weights of a neural network is as a *high dimensional shape*. As you train, this shape *molds* around the shape of your data, learning to distinguish one pattern from another. Unfortunately, the images in our testing dataset were *slightly* different than the patterns in our training dataset. This caused our network to fail on many of our *testing* examples.

This phenomenon is known as **Overfitting**. A more official definition of a neural network that overfits is a neural network that has learned the *noise* in the dataset instead of only making decisions based on the *true signal*.

Where Overfitting Comes From

What causes our neural networks to overfit?

Let's alter this scenario just a bit. Picture the fresh clay in your head again (unmolded). What if you only pushed a single fork into it? Assuming the clay was very thick, it wouldn't have as much detail as the previous mold did (which was in-printed many times) Thus, it would be only a *very general shape of a fork*. This shape might, in fact, be compatible with both the 3 and 4 pronged variety of fork, because it's still a very *fuzzy* imprint.

Assuming this information, our mold actually got worse at our testing dataset as we imprinted more forks because it learned more *detailed information* about the training dataset that it was being molded to. This caused it to reject images that were even the slightest bit off from what it had repeatedly seen in the training data. So, what is this *detailed information* in our images that is incompatible with our test data? In our fork analogy, this was the number of prongs on the fork. In images, it's generally referred to as *noise*. In reality, it's a bit more nuanced. Consider these two dog pictures.

Everything that makes these pictures unique *beyond* what captures the essence of "dog" is included in this term *noise*. In the picture on the left, the pillow and the background are both noise. In the picture on the right, the empty, middle blackness of the dog is actually a form of *noise* as well. It's really the edges that tell us that it's a dog. The middle blackness doesn't really tell us anything. On the picture on the left, however, the middle of the dog has the furry texture and color of a dog, which could help the classifier correctly identify it.

So, how do we get our neural networks to train only on the *signal* (the essence of a dog) and ignore the *noise* (other stuff irrelevant to the classification)? Well, one way of doing this is by **early stopping**. It turns out that a large amount of noise comes in the fine grained detail of an image, and most of the signal (for objects) is found in the general *shape* and perhaps *color* of the image.

The Simplest Regularization: Early Stopping

Stop training the network when it starts getting worse.

So, how do we get our neural networks to train only on the *signal* (the essence of a dog) and ignore the *noise* (other stuff irrelevant to the classification)? Well, one way of doing this is by **early stopping**. It turns out that a large amount of noise often comes in the fine grained detail present in input data, and most of the signal is found in the more general characteristics of your input data (for images, this is things like big shapes and color).

So, how do we get our neural network to ignore the fine grained detail and only capture the general information present in our data (i.e. the general shape of dog or of an MNIST digit)? Well, we don't let the network train long enough to learn it! Just like in the "fork mold" example, it takes many forks imprinted many times to create the perfect outline of a 3 pronged fork. The first few imprints only generally capture the shallow outline of a fork. The same can be said for neural networks. As a result, early stopping is the cheapest form of "regularization", and if you're in a pinch, it can be quite effective.

This brings us to the subject that this chapter is all about, **Regularization**. Regularization is a subfield of methods for getting your model to *generalize* to new data points (intead of just memorize the training data). It's a subset of methods that help your neural network learn the *signal* and ignore the *noise*. In our case, it's a toolset at our disposal to create neural networks that have these properties.

> ### Regularization
>
> A subset of methods used to encourage generalization in learned models, often by increasing the difficulty for a model to learn the fine-grained details of training data.

So, the next question might be, how do we know when to stop? In truth, the only real way to know is to run the model on data that isn't in your training dataset. This is typically done using a *second* test dataset called a "validation set". In some circumstances, if we used our test set for knowing when to stop, we could actually *overfit to our test set*. So, as a general rule, we don't use it to control training. We use a validation set instead.

You can see an example of validation being used in the github notebook "Chapter 6: Early Stopping", causing our previous net to stop at iteration 20.

Industry Standard Regularization: Dropout

The Method: randomly turning neurons off (setting to 0) during training.

So, this regularization technique really is as simple as it sounds. During training, you randomly set neurons in your network to zero (and usually the deltas on the same nodes during backpropagation, but you technically don't have to). This causes the neural network to train exclusively using *random subsections* of the neural network. Believe it or not, this regularization technique is generally accepted as the go-to, state-of-the-art regularization technique for the vast majority of networks. It's methodology is simple and inexpensive, although the intuitions behind *why* it works are a bit more complex.

> **Why does Dropout Work? (perhaps oversimplified)**
>
> Dropout makes our big network act like a little one by randomly training little subsections of the network at a time, and little networks don't overfit.

It turns out that the smaller a neural network is, the less it is able to overfit. Why? Well, small neural networks don't have very much expressive power. They can't latch onto the more granular details (i.e. noise) that tend to be the source of overfitting. They only have "room" to capture the big, obvious, high level features.

This notion of "room" or "capacity" is actually a really important one for you to keep in your mind. You can think of it like this. Remember our "clay" analogy from a few pages ago? Imagine if your clay was actually made of "sticky rocks" that were the size of dimes. Would that clay be able to make a very good imprint of a fork? Of course not! Why? Well, those stones are much like our weights. They form around our data, capturing the patterns we're interested in. If we only have a few, larger stones, then it can't capture nuanced detail. Each stone instead is pushed on by large parts of the fork, more or less *averaging* the shape (ignoring fine creases and corners).

Imagine again clay made up of very fine-grained sand. It's actually made up of millions and millions of small stones that can fit into every nook and cranny of a fork. This is what gives *big* neural networks the expressive power they often use to overfit to a dataset.

So, how do we have the power of a large neural network with the resistance to overfitting of the small neural network? We take our big neural network and turn off nodes randomly. What happens when you take a big neural network and only use a small part of it? Well, it behaves like a small neural network! However, when we do this randomly over potentaily millions of different "sub-networks", the sum total of the entire network still maintains its expressive power! Neat, eh?

Why Dropout Works: Ensembling Works

Dropout is actually a form of training a bunch of networks and averaging them

Something to keep in mind: neural networks always start out randomly. Why does this matter? Well, since neural networks learn by trial and error, this ultimately means that every neural network learns just a little bit *differently*. It might learn equally effectively, but no two neural networks are ever exactly the same (unless they start out exactly the same for some random or intentional reason).

This has an interesting property. When you overfit two neural networks, no two neural networks overfit in exactly the same way. Why? Well, overfitting only occurs until every training image can be predicted perfectly, at which point the error == 0 and the network stops learning (even if you keep iterating). However, since each neural network starts by predicting randomly, then adjusting its weights to make better predictions, each network inevitably makes different mistakes, resulting in different updates. This culminates in a core concept.

> While it is very likely for large, unregularized neural networks to overfit to noise, it is very *unlikely* for them to overfit to the **same** noise.

Why do they not overfit to the same noise? Well, they start randomly, and they stop training once they have learned enough noise to disambiguate between all the images in the training set. Truth be told, our MNIST network only needs to find a handful of random pixels that happen to correlate with our output labels to overfit. However, this is contrasted with, perhaps, an even more important concept.

> Neural networks, even tough they are randomly generated, still start by learning the biggest, most broad sweeping features before learning much about the noise.

The takeaway is this; if you train 100 neural networks (all initialized randomly), they will each tend to latch onto different noise but similar broad *signal*. Thus, when they make mistakes, they often make *differing* mistakes. This means that if we allowed them to vote equally, their noise would tend to cancel out, revealing only what they have all learned in common, *the signal*.

Dropout In Code

Here's how you actually use Dropout in the real world

In our MNIST classification model, we're going to add Dropout to our hidden layer, such that 50% of the nodes are turned off (randomly) during training. Perhaps you will be surprised that this is actually only a 3 line change in our code. Below you can see a familiar snippet from our previous neural network logic with our dropout mask added.

```
layer_0 = images[i:i+1]
dropout_mask = np.random.randint(2,size=layer_1.shape)

layer_1 *= dropout_mask * 2
layer_2 = np.dot(layer_1, weights_1_2)

error += np.sum((labels[i:i+1] - layer_2) ** 2)

correct_cnt += int(np.argmax(layer_2) == \
          np.argmax(labels[i+i+1]))

layer_2_delta = (labels[i:i+1] - layer_2)
layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
          * relu2deriv(layer_1)

layer_1_delta *= dropout_mask

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

Indeed, to implement Dropout on a layer (in this case layer_1), one simply multiplies the layer_1 values by a random matrix of 1s and 0s. This has the affect of randomly "turning off" nodes in layer_1 by setting them to equal 0. Note that our dropout_mask uses what's called a "50% bernoulli distribution" such that 50% of the time, each value in the dropot_mask is a 1, and (1 - 50% = 50%) of the time, it is a 0.

This is followed by something that may seem a bit peculiar. We multiply layer_1 by two! Why do we do this? Well, remember that layer_2 is going to perform a *weighted sum* of layer_1. Even though it's weighted, it's still a **sum** over the values of layer_1. Thus, if we turn off half the nodes in layer_1, then that sum is going to be cut in half! Thus, layer_2 would increase its sensitivity to layer_2, kindof like a person leaning closer to a radio when the volume is too low to better hear it. However, at test time, when we no longer use Dropout, the volume would be back up to normal! You may be surprised to find that this throws off layer_2's ability to *listen* to layer_1. Thus, we need to counter this by multiplying layer_1 by (1 / the percentage of turned on nodes). In this case, that's 1/0.5 which equals 2. In this way, the volume of layer_1 is the same between training and testing, despite Dropout.

```
import numpy, sys
np.random.seed(1)
def relu(x):
    return (x >= 0) * x # returns x if x > 0
                        # returns 0 otherwise

def relu2deriv(output):
    return output >= 0 #returns 1 for input > 0

alpha, iterations, hidden_size = (0.005, 300, 100)
pixels_per_image, num_labels = (784, 10)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0,0)
    for i in range(len(images)):
        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2, size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[i:i+1] - layer_2) ** 2)
        correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(labels[i:i+1]))
        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1)
        layer_1_delta *= dropout_mask

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

    if(j%10 == 0):
        test_error = 0.0
        test_correct_cnt = 0

        for i in range(len(test_images)):
            layer_0 = test_images[i:i+1]
            layer_1 = relu(np.dot(layer_0,weights_0_1))
            layer_2 = np.dot(layer_1, weights_1_2)

            test_error += np.sum((test_labels[i:i+1] - layer_2) ** 2)
            test_correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(test_labels[i:i+1]))

        sys.stdout.write("\n" + \
            "I:" + str(j) + \
            " Test-Err:" + str(test_error/ float(len(test_images)))[0:5] +\
            " Test-Acc:" + str(test_correct_cnt/ float(len(test_images)))+\
            " Train-Err:" + str(error/ float(len(images)))[0:5] +\
            " Train-Acc:" + str(correct_cnt/ float(len(images))))
```

Dropout Evaluated on MNIST

Here's how you actually use Dropout in the real world.

If you remember from before, our neural network (without Dropout) previously reached a test accuracy of 75.9% before falling down to finish training at 62.3% accuracy. When we add dropout, our neural network instead behaves this way.

```
I:0 Test-Err:0.681 Test-Acc:0.57 Train-Err:0.926 Train-Acc:0.388
I:10 Test-Err:0.495 Test-Acc:0.746 Train-Err:0.474 Train-Acc:0.769
I:20 Test-Err:0.454 Test-Acc:0.77 Train-Err:0.429 Train-Acc:0.824
I:30 Test-Err:0.460 Test-Acc:0.762 Train-Err:0.402 Train-Acc:0.835
I:40 Test-Err:0.467 Test-Acc:0.775 Train-Err:0.392 Train-Acc:0.836
I:50 Test-Err:0.460 Test-Acc:0.773 Train-Err:0.381 Train-Acc:0.843
I:60 Test-Err:0.458 Test-Acc:0.777 Train-Err:0.362 Train-Acc:0.869
I:70 Test-Err:0.466 Test-Acc:0.758 Train-Err:0.356 Train-Acc:0.879
I:80 Test-Err:0.454 Test-Acc:0.764 Train-Err:0.348 Train-Acc:0.879
I:90 Test-Err:0.454 Test-Acc:0.758 Train-Err:0.332 Train-Acc:0.895
I:100 Test-Err:0.452 Test-Acc:0.754 Train-Err:0.311 Train-Acc:0.905
I:110 Test-Err:0.456 Test-Acc:0.754 Train-Err:0.316 Train-Acc:0.903
I:120 Test-Err:0.450 Test-Acc:0.753 Train-Err:0.312 Train-Acc:0.903
I:130 Test-Err:0.450 Test-Acc:0.772 Train-Err:0.292 Train-Acc:0.913
I:140 Test-Err:0.455 Test-Acc:0.776 Train-Err:0.306 Train-Acc:0.918
I:150 Test-Err:0.456 Test-Acc:0.772 Train-Err:0.289 Train-Acc:0.927
I:160 Test-Err:0.464 Test-Acc:0.765 Train-Err:0.295 Train-Acc:0.917
I:170 Test-Err:0.462 Test-Acc:0.762 Train-Err:0.284 Train-Acc:0.928
I:180 Test-Err:0.466 Test-Acc:0.764 Train-Err:0.282 Train-Acc:0.922
I:190 Test-Err:0.468 Test-Acc:0.766 Train-Err:0.274 Train-Acc:0.934
I:200 Test-Err:0.472 Test-Acc:0.762 Train-Err:0.283 Train-Acc:0.94
I:210 Test-Err:0.478 Test-Acc:0.753 Train-Err:0.277 Train-Acc:0.93
I:220 Test-Err:0.471 Test-Acc:0.77 Train-Err:0.263 Train-Acc:0.938
I:230 Test-Err:0.474 Test-Acc:0.772 Train-Err:0.260 Train-Acc:0.937
I:240 Test-Err:0.488 Test-Acc:0.759 Train-Err:0.272 Train-Acc:0.935
I:250 Test-Err:0.480 Test-Acc:0.778 Train-Err:0.258 Train-Acc:0.939
I:260 Test-Err:0.488 Test-Acc:0.768 Train-Err:0.262 Train-Acc:0.943
I:270 Test-Err:0.487 Test-Acc:0.769 Train-Err:0.254 Train-Acc:0.945
I:280 Test-Err:0.503 Test-Acc:0.772 Train-Err:0.262 Train-Acc:0.938
I:290 Test-Err:0.509 Test-Acc:0.77 Train-Err:0.247 Train-Acc:0.949
```

Not only does the network instead peak out at a score of 77.8%, it also doesn't over fit nearly as badly, finishing training with a testing accuracy of 77%. Notice that the dropout also slows down the Training-Acc, which previously went straight to 100% and then stayed there.

This should point to what Dropout really is. It's noise. It makes it more difficult for the network to train on the training data. It's like running a marathon with weights on your legs. It's harder to train, but when you take them off for the big race, you end up running quite a bit faster becasue you trained for something that was much more difficult.

Batch Gradient Descent

A method for increasing the speed of training and the rate of convergence.

In the context of this chapter, I would like to briefly apply a concept introduced several chapters ago, the concept of mini-batched stochastic gradient descent. I won't go into too much detail, as it's something that's largely taken for granted in neural newtork training. Furthemore, it's a very simple concept that doesn't really get more advanced even with the most state of the art neural networks. Simply stated, previously we trained one training example at a time, updating the weights after each example. Now, we're going to train 100 training examples at a time averaging the weight updates between all 100 examples. The code for this training logic is on the next page, and the training/testing output is below.

```
I:0 Test-Err:0.849 Test-Acc:0.325 Train-Err:1.347 Train-Acc:0.159
I:10 Test-Err:0.603 Test-Acc:0.643 Train-Err:0.602 Train-Acc:0.659
I:20 Test-Err:0.547 Test-Acc:0.708 Train-Err:0.538 Train-Acc:0.726
I:30 Test-Err:0.522 Test-Acc:0.735 Train-Err:0.503 Train-Acc:0.745
I:40 Test-Err:0.502 Test-Acc:0.746 Train-Err:0.487 Train-Acc:0.746
I:50 Test-Err:0.491 Test-Acc:0.752 Train-Err:0.466 Train-Acc:0.791
I:60 Test-Err:0.485 Test-Acc:0.762 Train-Err:0.447 Train-Acc:0.789
I:70 Test-Err:0.474 Test-Acc:0.767 Train-Err:0.439 Train-Acc:0.791
I:80 Test-Err:0.473 Test-Acc:0.769 Train-Err:0.441 Train-Acc:0.794
I:90 Test-Err:0.470 Test-Acc:0.759 Train-Err:0.439 Train-Acc:0.804
I:100 Test-Err:0.470 Test-Acc:0.756 Train-Err:0.419 Train-Acc:0.815
I:110 Test-Err:0.465 Test-Acc:0.758 Train-Err:0.413 Train-Acc:0.828
I:120 Test-Err:0.467 Test-Acc:0.769 Train-Err:0.407 Train-Acc:0.814
I:130 Test-Err:0.465 Test-Acc:0.773 Train-Err:0.403 Train-Acc:0.818
I:140 Test-Err:0.469 Test-Acc:0.768 Train-Err:0.413 Train-Acc:0.835
I:150 Test-Err:0.469 Test-Acc:0.771 Train-Err:0.397 Train-Acc:0.845
I:160 Test-Err:0.469 Test-Acc:0.776 Train-Err:0.403 Train-Acc:0.84
I:170 Test-Err:0.471 Test-Acc:0.772 Train-Err:0.394 Train-Acc:0.859
I:180 Test-Err:0.468 Test-Acc:0.775 Train-Err:0.391 Train-Acc:0.84
I:190 Test-Err:0.466 Test-Acc:0.78 Train-Err:0.374 Train-Acc:0.859
I:200 Test-Err:0.469 Test-Acc:0.783 Train-Err:0.392 Train-Acc:0.862
I:210 Test-Err:0.470 Test-Acc:0.769 Train-Err:0.378 Train-Acc:0.861
I:220 Test-Err:0.466 Test-Acc:0.782 Train-Err:0.369 Train-Acc:0.864
I:230 Test-Err:0.465 Test-Acc:0.786 Train-Err:0.368 Train-Acc:0.87
I:240 Test-Err:0.469 Test-Acc:0.782 Train-Err:0.369 Train-Acc:0.869
I:250 Test-Err:0.470 Test-Acc:0.776 Train-Err:0.362 Train-Acc:0.876
I:260 Test-Err:0.466 Test-Acc:0.787 Train-Err:0.371 Train-Acc:0.859
I:270 Test-Err:0.469 Test-Acc:0.778 Train-Err:0.357 Train-Acc:0.877
I:280 Test-Err:0.469 Test-Acc:0.778 Train-Err:0.371 Train-Acc:0.878
I:290 Test-Err:0.466 Test-Acc:0.775 Train-Err:0.359 Train-Acc:0.873
```

Notice that our training accuracy has a bit of a smoother trend to it than it did before. Furthermore, we actually reach a slightly higher testing accuracy of 78.7%! Taking an average weight update consistently creates this kind of phenomenon during training. As it turns out, individual training examples are very noisy in terms of the weight updates they generate. Thus, averaging them makes for a smoother learning process.

```python
import numpy as np
np.random.seed(1)

def relu(x):
    return (x >= 0) * x # returns x if x > 0

def relu2deriv(output):
    return output >= 0 # returns 1 for input > 0


alpha, iterations = (0.1, 300)
pixels_per_image, num_labels, hidden_size = (784, 10, 100)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0, 0)
    for i in xrange(len(images) / batch_size):
        batch_start, batch_end = ((i * batch_size),((i+1)*batch_size))

        layer_0 = images[batch_start:batch_end]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2,size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[batch_start:batch_end] - layer_2) ** 2)
        for k in xrange(batch_size):
            correct_cnt += int(np.argmax(layer_2[k:k+1]) == \
                            np.argmax(labels[batch_start+k:batch_start+k+1]))

        layer_2_delta = (labels[batch_start:batch_end]-layer_2)/batch_size
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
            * relu2deriv(layer_1)
        layer_1_delta *= dropout_mask

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

The first thing you'll notice when running this code is that it runs way faster. This is because each "np.dot" function is now performing 100 vector dot products at a time. As it turns out, CPU architectures are way faster at performing dot products batched in this way.

There's actually more going on here, however. Notice that our alpha is 20x larger than it ways before. We can increase this for a rather fascinating reason. Consider if you were trying to find a city using a very wobbly compass. If you just looked down, got a heading, and then ran 2 miles, you'd likely be way off course! However, if you looked down, took 100 headings and then averaged them, running 2 miles would probably take you in the general right direction. Thus, because we're taking an average of a noisy signal (i.e., the average weight change over 100 training examples), we can take bigger steps! You will generally see batching ranging from size 8 to as high as 256. Generally, researchers pick numbers randomly until they find a batch_size/alpha pair that seems to work well.

Conclusion

In this chapter we have addressed two of the most widely used methods for increasing the accuracy and training speed of almost any neural architecture. In the following chapters, we will pivot from sets of tools that are universally applicable to nearly all neural networks to special purpose architectures that are advantageous for modeling specific types of phenomeon in data. See you there!

Modeling Probabilities and Non-Linearities
Activation Functions

9

IN THIS CHAPTER ··

> 66
>
> 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 is an Activation Function?

Definition: a function applied to the neurons in a layer during prediction.

An activation function is a function applied to the neurons in a layer during prediction. This should seem very familiar to you, as we have already been using an activation function called "relu" (pictured right in our 3 layer neural network). The relu function had the afffect of turning all negative numbers to 0.

First and foremost, an activation function is any function that can take one number and return another number. However, there are an infinite number of functions in the universe, and not all of them are useful as activation functions. There are several constraints on what makes a function an "Activation Function". Using functions outside of these constraints is usually a bad idea, as we will see.
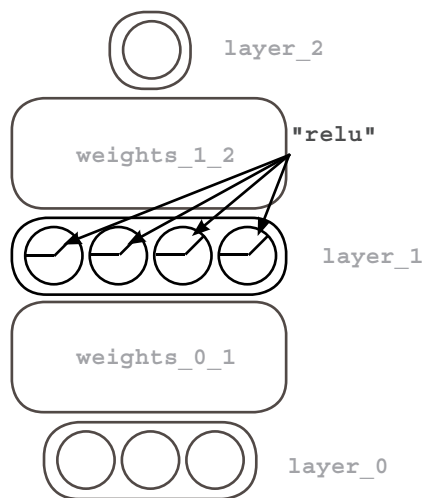
Constraint 1: the function must be continuous and infinite in domain.

The first constraint on what makes a proper "Activation Function" is that it must have an output number for *any* input. In other words, we shouldn't be able to put in a number that doesn't have an output for some reason. A bit overkill, but see how the function on the left (4 distinct lines) doesn't have y values for every x value? It's only defined in 4 spots. This would make for a horrible activation function. The function on the right, however, is continuous and infinite in domain. There is no input (x) for which we can't compute an output(y).

$(y = x * x)$
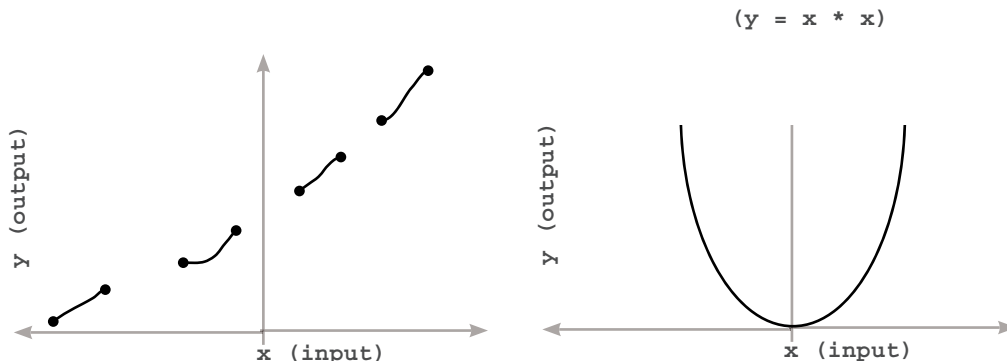
Constraint 2: good activation functions are "monotonic", never changing direction.

The second constraint is that the function is 1:1. It must never "change direction". In other words, it must either be "always increasing" or "always decreasing". As an example look at the two functions below. What these shapes show is, given x as input, what value of y does the function describe? The function on the left (y = x * x) cannot be used as an activation function because it is not either "always increasing" or "always decreasing". How can we tell? Well, notice that there are many cases in which two values of x have a single value of y (this is true for every value except 0, actually). The function on the right, however, is always increasing! There is no point at which 2 values of x have the same value of y.
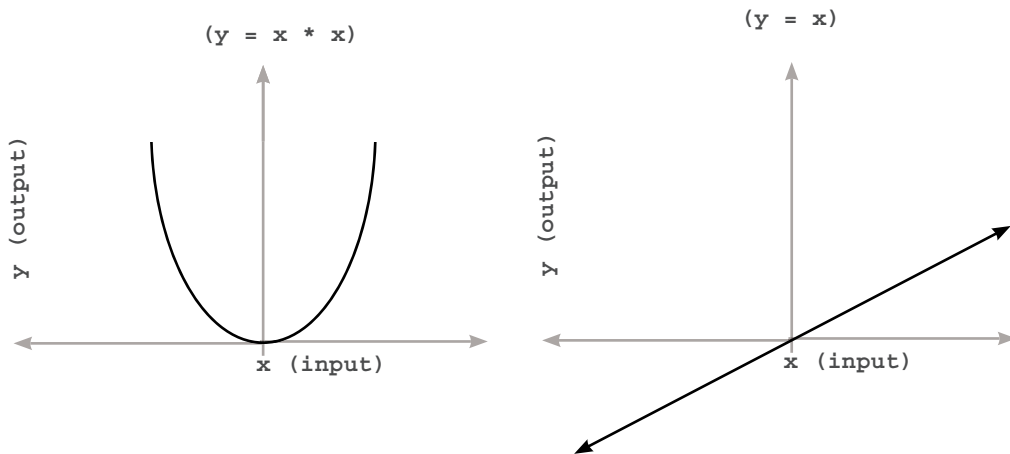
Now, this particular constraint is not technically a "requirement". Unlike functions that have missing values (non-continuous), we can optimize functions that aren't monotonic. However, consider the implication of having multiple input values map to the same output value. When we're learning in our neural networks, we're "searching" for the right weight configurations to give us a specific output. This problem can get a lot harder if we have mulitple "right answers". In other words, if there are multiple ways to get the same output, then the network has multiple possible "perfect" configurations.

An optimist might say, "Hey, this is great! We're more likely to find the right answer if it can be found in multiple places!." A pessimist would say, "This is terrible! Now we don't have a 'correct direction to go to reduce the error' since we can go in either direction and theoretically make progress." Unfortunately, the phenomenon the pessimist identified is more important. For an advanced study of this subject, look more into "Convex vs Non-Convex Optimization". Many universities (and online courses) will have entire courses dedicated to these kinds of questions.

Constraint 3: Good activation functions are "nonlinear" (i.e., they squiggle or turn)

The third constraint requires a bit of recollection back to our chapter on deep neural networks. Remember how we needed to have "Sometimes Correlation"? In order to create this "Sometimes Correlation", we needed to allow our neurons to "selectively correlate to input neurons" such that a really negative signal from one input into a neuron could reduce how much it correlated to any input at all (by forcing the neuron to simply drop to 0, in the case of "relu").

As it turns out, this phenomenon is facilitated by *any function that curves*. Functions that look like "straight lines", on the other hand, simply scale the weighted average coming in. Simply scaling something (multiplying it by a constant... like "2") doesn't affect how correlated a neuron is to its various inputs. It simply makes the collective correlation that is there louder or softer. However, the activation doesn't allow one weight to affect how correlated the neuron is to the other weights. What we *really* want, is *selective* correlation. Given a neuron with an activation function, we want one incoming signal to be able to increase or decrease how correlated the neuron is to all the other incoming signals. All curved lines do this (to varying degrees, as we will see).

Thus, the function on the left would be considered a "linear" function, whereas the one on the right is considered "non-linear" and will usually make for a better activation function (there are exceptions that we'll discuss later).
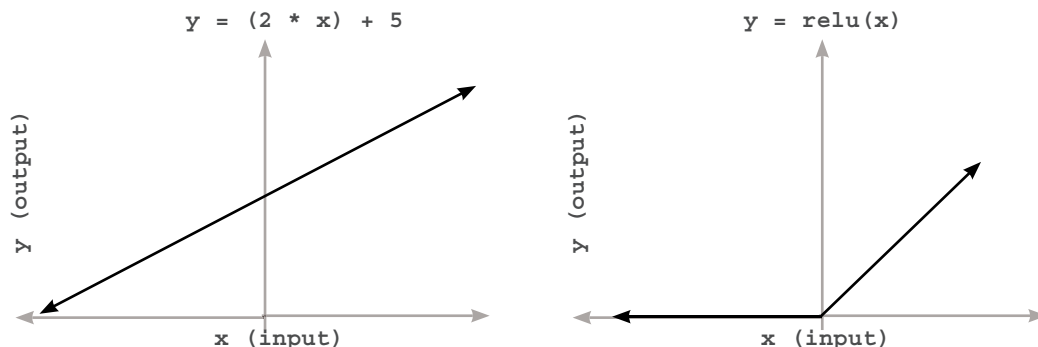
Constraint 4: Good activation functions (and their derivatives) should be efficiently computable.

This one is pretty simple. You're going to be calling this function a lot (sometimes billions of times), so you don't want it to be too slow to compute. In fact, many recent activation functions have become popular because they are so easy to compute at the expense of their expressiveness ("relu" is a great example of this).

Standard Hidden Layer Activation Functions

Out of the infinite possible functions, which ones are most commonly used?

Even with these constraints, it should be clear that there are an infinite (possibly transfinite?) number of functions that could be used as activation functions. In fact, the last few years have seen a lot of progress in state-of-the-art activations. However, there are still a relatively small list of activations that account for the vast majority of activation needs, and improvements on them have been relatively minute in most cases.

Sigmoid: The Bread and Butter

Sigmoid is great because it smoothly squishes the infinite amount of input to an output between 0 and 1. In many circumstances, this lets you interpret the output of any individual neuron as a "probability". Thus, people use this non-linearity both in hidden layers and output layers. (image: wikipedia)
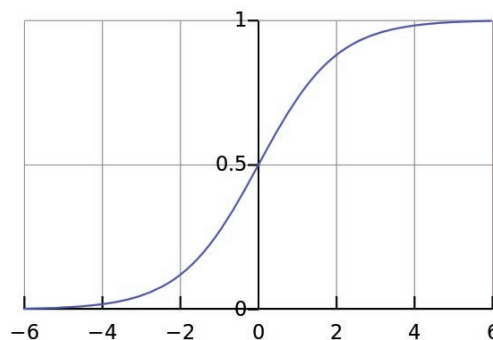
Tanh: better than Sigmoid for Hidden

Here's the cool thing about Tanh. Remember how we wanted to model "selective correlation?" Well, Sigmoid gives you "varying degrees of positive correlation". That's nice. Tanh is the same as sigmoid *except* it's between -1 and 1! This means it can also throw in some *negative correlation*. While not that useful for output layers (unless the data you're predicting goes between -1 and 1), this aspect of *negative correlation* is very powerful for hidden layers and on many problems, tanh will outperform sigmoid in hidden layers. (image: wolfram alpha)
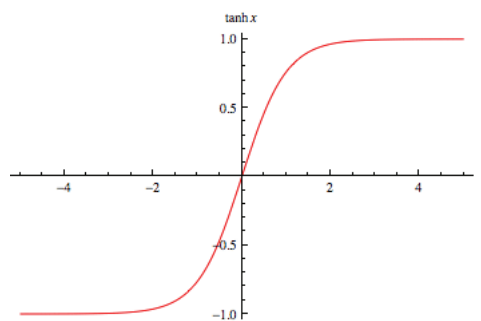
Standard Output Layer Activation Functions

Choosing the best one depends on what you're trying to predict.

So, it turns out that what is best for Hidden Layer activation functions can be quite different than what is best for Output Layer activation functions, especially when it comes to classification. Broadly speaking, there are 3 major types of output layer.

Configuration 1: Predicting Raw Data Values (No Activation Function)

This is perhaps the most straightforward but least common type of output layer. In some cases, people want to train a neural network to transform "one matrix of numbers" into "another matrix of numbers", where the range of the output (difference between lowest and highest value) is something other than a probability. One example might be predicting the average temperature in Colorado given the tempreature in the surrounding states.

The main thing to focus on here is to ensure that your output non-linearity can actually predict the right answers. In the case above, a sigmoid or tanh would be inappropriate since they force every prediction to be between 0 and 1 (but we want it to predict any temperature, not just between 0 and 1). If I were training a network to do this prediction, I would very likely just train the network without an activation function on the output at all.

Configuration 2: Predicting Unrelated Yes/No Probabilities (Sigmoid)

There will often be times where you want to make multiple binary probabilities in one neural network. We actually already did this in our "Networks with Multiple Inputs and Outputs" subchapter, where we predicted whether the team would win, whether there would be injuries, and the morale of the team (happy or sad) based on the input data. As an aside, when you have neural networks with hidden layers, predicting multiple things at once can be beneficial. Often times the network will learn something in predicting one label that will be useful to one of your other labels. For example, if the network got really good at predicting whether or not the team would win ballgames, the same hidden layer would likely be very useful for predicting whether the team would be happy or sad. However, the network might have a harder time simply predicting happiness or sadness without this extra signal. This tends to vary greatly from problem to problem, but it's good to keep in mind.

In these instances, it's best to use the Sigmoid activation function as it models individual probabilities separately for each output node.

Configuration 3: Predicting "Which One" Probabilities (Softmax)

By far the most common usecase in neural networks is when you want to predict a single label out of many. For example, in our MNIST digit classifier, we want to predict *which* number is in the image. We know ahead of time that the image cannot be more than one number at a time. We could train this network with a sigmoid activation function and simply declare that the highest output probability is the most likely! This will work reasonably well. However, it is far better to have an activation function that models the idea that "The more likely it's one label, the less likely it is any of the other labels."

Why do we like this phenomenon? Well, consider how weight updates are performed. Let's say our MNIST digit classifier should predict that the image is a 9. Let's say that the raw weighted sums going into our final layer (before we apply an activation function) are the following values.

```
                  0      1      2      3      4      5      6      7      8      9
raw dot
product         (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (100)
 values
```

In other words, the network's raw input to the last layer predicts a "0" for every node but 9, where it predics 100. We might call this "perfect". Let's see what happens when we run these numbers through a sigmoid activation function.

```
sigmoid         (.50)  (.50)  (.50)  (.50)  (.50)  (.50)  (.50)  (.50)  (.50)  (.99)
```

Strangely, the network seems less sure now! I mean, 9 is still the highest, but it seems to think that there's a 50% chance that it could have been any one of the other numbers as well. Wierd! Softmax, on the other hand, would interpret the input very differently.

```
softmax         (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (1.0)
```

This looks great! Not only is 9 the highest, but the network doesn't even suspect it's any of the other possible MNIST digits! This might just seem like just a theoretical flaw of sigmoid, but it can have serious consequences when we backpropagate. Consider how our "Mean Squared Error" is calculated on the *sigmoid* output. In theory, the network is predicting nearly perfectly, right? Surely it won't backprop hardly any error. Not so for sigmoid!

```
sigmoid         (.25)  (.25)  (.25)  (.25)  (.25)  (.25)  (.25)  (.25)  (.25)  (.00)
  MSE
```

Look at all of the error! These weights are in for a *massive* weight update even though the network predicted perfectly! Why? Well, for sigmoid to reach 0 error, it doesn't just have to predict the highest positive number for the true output, but it has to predict a 0 everywhere else. In other words, where softmax asks "which digit seems like the best fit for this input". Sigmoid says "you better believe that it's only digit "9" and doesn't have anything in common with the other MNIST digits!!".

The Core Issue: Inputs Have Similarity

Different numbers share characteristics. It's good to let the network believe that.

So, it turns out that MNIST digits aren't all completely different. In other words, they have overlapping pixel values! The average 2 shares quite a bit in common with the average 3. Why is this important? Well, as a general rule, similar inputs create similar outputs. If you take some numbers and multiply them by a couple matrices. If the starting numbers are pretty similar, the ending numbers are going to be pretty similar.

similar strokes!
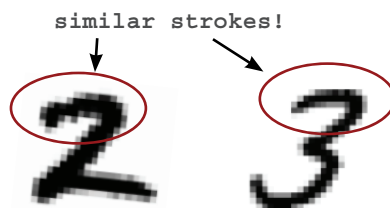
Consider the 2 and 3 above. If you forward propagate the 2 and a small amount of probability accidentally goes to the label "3", what does it mean for the network to consider this a *big mistake* and respond with a *big weight update*? It means that it's going to penalize the network for recognizing a 2 by anything other than features that are exclusively related to 2s! It penalizes the network for recognizing a 2 based on... say... the top curve! Why? Well, 2 and 3 share the same curve at the top of the image. Training with a sigmoid would penalize the network for trying to predict a 2 based on this input, because by doing so it would

be looking for the same input as it does for 3s. Thus, when a 3 came along, the "2" label would get some probability (because part of the image looks "2ish").

What's the side affect? Well, since most images share lots of pixels in the middle of images, the network will start trying to focus on the edges of the images. Consider this "2 detector" node's weights. See how muddy the middle of the image is? The heaviest weights are the end points of the 2 that are toward the edge of the image. On one hand, these are probably

the *best individual* indicators of a 2, but the best overall is going to be a newtork that sees the entire shape for what it is. These individual indicators can be accidentally triggered by a 3 that is slightly off center or tilted in the wrong way. It's not learning the true essence of a 2 because it needs to learn "2 and NOT 1 NOT 3, NOT 4, etc..,".
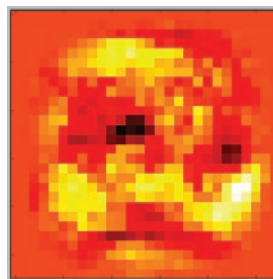
What we really want is an output activation that won't penalize labels that are similar. Instead, we want it to pay attention to all the information that can be indicative of any potential input. Furthermore, it's also quite nice that a softmax's probabilities always sum to 1. We can interpret any individual prediction as a global probability that the prediction is a particular label. Softmax works better in both theory and practice.
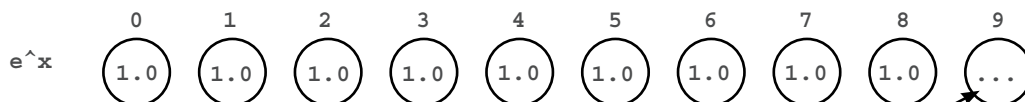
Softmax Computation

Softmax raises each input value exponentially and then divides by the layer's sum.

Let's see a softmax computation on our neural network's hypothetical output values from a few pages ago. I'll picture them here again so we can see the input to softmax.

```
               0      1      2      3      4      5      6      7      8      9
raw dot
product     (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (100)
 values
```

      The first thing that you must do to compute a softmax on the whole layer is to raise each value exponentially. So for each value x, we are going to compute e to the power of x. (e is a special number ~2.71828...). You can see the value of e^x on the right. Notice that it turns every prediction into a positive number, where negative numbers turn into *very small* positive numbers and big numbers turn into *very big* numbers. (If you've heard of "exponential growth" it was likely talking about this function or one very similar to it.)

```
               0      1      2      3      4      5      6      7      8      9
 e^x        (1.0)  (1.0)  (1.0)  (1.0)  (1.0)  (1.0)  (1.0)  (1.0)  (1.0)  (...)

                                                               2.688 * 10^43
```
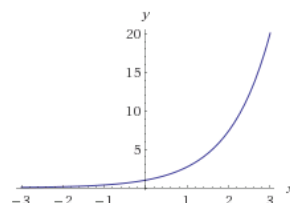
      In short, all the 0s turned to 1s (Because 1 is the "y intercept" of e^x) and the 100 turned in to a massive number (2 followed by 43 zeros!). If there were any negative numbers, they would have turned into something between 0 and 1. The next step is to sum all the nodes in the layer, and divide each value in the layer by that sum. This effectively makes every number 0 except for the value for label 9.

```
softmax    (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (0.0)  (1.0)
```

      So, what is it about softmax that we like? The nice thing about softmax is that the higher the network predicts one value, the lower it predicts all the others. It increases what is called "sharpness of attenuation". It encourages the network to predict one output with very high probability.

      If you wanted to adjust how aggressively it does this, you could simply use numbers slightly higher or lower than "e" when you're exponentiating. Lower numbers will result in lower attenuation, and higher numbers will result in higher attenuation. However, most people just stick with "e".

Activation Installation Instructions

How do you add your favorite activation function to any layer?

Now that we've covered a wide variety of activation functions and explained their usefulness in hidden and output layers of neural networks, let's talk about the proper way to install one into a neural network. Fortunately, we've already seen an example of how to use a non-linearity in our first deep neural network. In this case, we added a "relu" activation function to our hidden layer. Adding this to forward propagation was relatively straightforward. We just took what layer_1 would have been (without an activation) and applied the "relu" function to each value.

```
layer_0 = images[i:i+1]
layer_1 = relu(np.dot(layer_0,weights_0_1))
layer_2 = np.dot(layer_1,weights_1_2)
```

There's a bit of lingo here to remember. The "input" to a layer refers to the value before the non-linearity. In this case, the input to layer_1 is `np.dot(layer_0,weights_0_1)`. This is not to be confused with the "previous layer" itself, layer_0.

Adding an activation function to a layer in forward propagation is relatively straightforward. However, properly compensating for the activation function in backpropagation is a bit more nuanced. If you remember back to Chapter 6, we performed an interesting operation to create the layer_1_delta variable. Wherever relu had forced a layer_1 value to be 0, we also multipled the delta by 0. Why did we do this? The reasoning at the time was, "Because a layer_1 value of 0 had no affect on the output prediction, it shouldn't have any impact on the weight update either. It wasn't responsible for the error." This is the extreme form of a more nuanced property. Consider the shape of the "relu" function.

The "slope" of relu for positive numbers is exactly 1. The "slope" of relu for negative numbers is exactly 0. What this really means is that modifying the input to this function (by a tiny amount) will have a 1:1 affect if it was predicting positively, and will have a 0:1 (none) affect if predicting negatively. This "slope" is a measure of how much the output of relu will change given a change in its input. Since the point of the "delta" at this point is to tell earlier layers "make my input higher or lower next time", this delta is very useful! It modifies the delta backpropagated from the following layer to take into account whether this node contributed to the error.

input higher or lower next time", this delta is very useful! It modifies the delta backpropagated from the following layer to take into account whether this node contributed to the error.
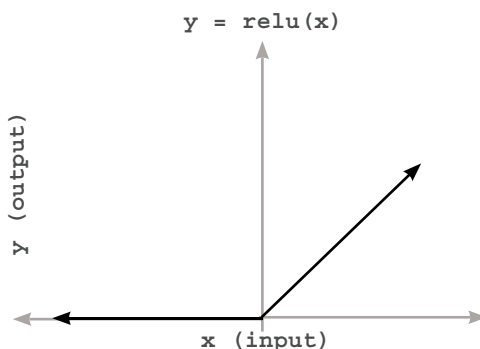
Thus, when we backpropagate, in order to generate layer_1_delta, we multiply the backpropagated delta from layer_2 (`layer_2_delta.dot(weights_1_2.T)`) by the slope of relu *at the point predicted in forward propagation*. For some deltas the slope is 1 (positive numbers) and others is is 0 (negative numbers).

```
error += np.sum((labels[i:i+1] - layer_2) ** 2)

correct_cnt += int(np.argmax(layer_2) == \
                                np.argmax(labels[i:i+1]))

layer_2_delta = (labels[i:i+1] - layer_2)
layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
                        * relu2deriv(layer_1)

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


def relu(x):
    return (x >= 0) * x # returns x if x > 0
                        # return 0 otherwise

def relu2deriv(output):
    return output >= 0 # returns 1 for input > 0
                       # return 0 otherwise
```

"relu2deriv" is a special function that can take the output of "relu" and calculate the slope of "relu" at that point (and it does this for all the values in the "output" vector). This begs the question, how do we make similar adjustments for all of the other non-linearities that aren't "relu". Consider "relu" and "sigmoid"
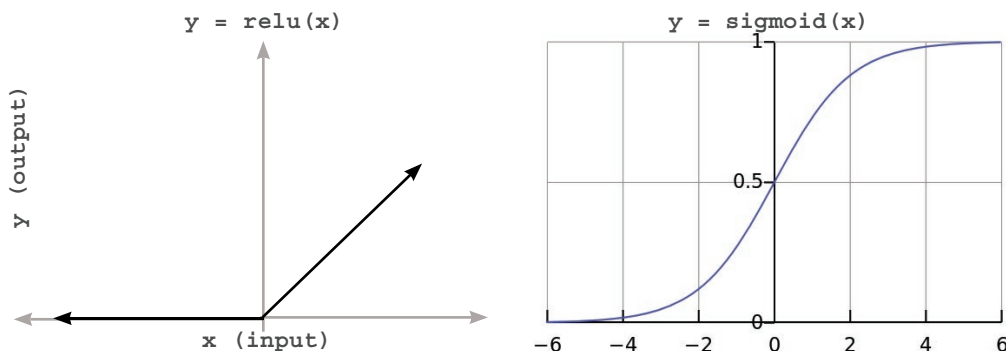
The important point to see in these pictures is that the "slope" is an indicator of how much a *tiny* change to the input would affect the output. We want to modify our incoming delta (from the following layer) to take into account whether or not a weight update before this node would have any affect. Remember, the end goal is to adjust weights to reduce our error. This steps encourages the network to leave weights alone if adjusting them will have little to no affect. It does this by multiplying it by the slope. This is no different for sigmoid.

Multiplying Delta By The Slope

To compute layer_delta, we multiply the backpropagated delta by the layer's slope

layer_1_delta[0] represents how much higher or lower the first hidden node of layer 1 should be in order to reduce the error of the network (for a particular training example). When there is no non-linearity, this is simply the weighted average delta of layer_2. However, the end goal of the delta on a neuron is to inform the weights before it about whether or not they should move. If moving them would have no affect, they (as a group) should be left alone. This is obvious for "relu" which is either "on" or "off". Sigmoid is, perhaps, more nuanced.
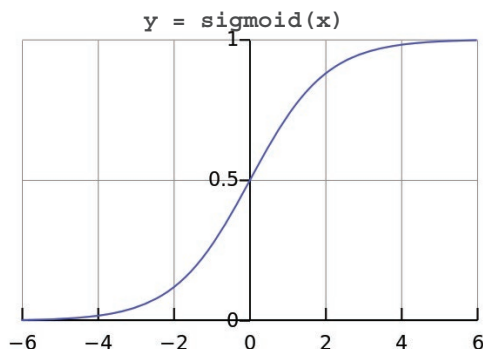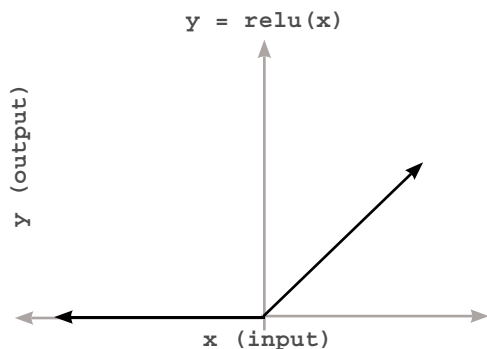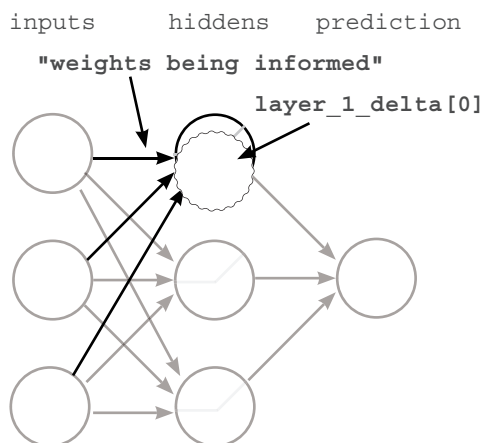
Consider a single sigmoid neuron. Sigmoid's "sensitivity to change in the input" slowly increases as the input approaches 0 from either direction. However, very positive and very negative inputs approach a slope of very near 0. Thus, as the input becomes very positive or very negative, small changes to the incoming weights become less relevant to the neuron's error at this training example. In broader terms, many hidden nodes are irrelevant to the accurate prediction of a "2" (perhaps they're only used for "8"s). Thus, we shouldn't mess with their weights too much becasue we could corrupt their usefulness elsewhere.

Inversely, this also creates a notion of "stickiness". Weights that have previously been updated a lot in one direction (for similar training examples) are very "confident"-ly predicting a high value or low value. These non-linearities help make it harder for occasional erroneous training examples to corrupt intelligence that has been reinforced many times.

Converting Output to Slope (derivative)

Most great activations can convert their output to their slope (efficiency win!)

So, now that we see that adding an activation to a layer changes how we compute the delta for that layer, we should discuss how the industry does this efficiently. The new operation that is necessary is the computation of the derivative of whatever nonlinearity was used.

Most nonlinearities (all the popular ones?) utilize a method of computing a derivative that will seem surprising to those of you who are already familiar with Calculus. Instead of computing the derivative at a certain point on its curve the "normal" way, most great activation functions have a means by which the *output* of the layer (at forward propagation) can be used to compute the derivative. This has become the standard practice for computing derivatives in neural networks and it is quite handy. Below is a small table for the functions we've seen so far, paired with their derivatives. `input` is a numpy vector (corresponding to the input to a layer). `output` is the prediction of the layer. `deriv` is the derivative of the vector of activation derivatives corresponding to the derivaitve of the activation at each node. `true` is the vector of true values (typically a 1 for the correct label position, 0 everywhere else).

function	forward prop	back prop delta
relu	`ones_and_zeros = (input > 0)` `output = input*ones_and_zeros`	`mask = output > 0` `deriv = output * mask`
sigmoid	`output = 1/(1 + np.exp(-input))`	`deriv = output*(1-output)`
tanh	`output = np.tanh(input)`	`deriv = 1 - (output**2)`
softmax	`temp = np.exp(input)` `output /= np.sum(temp)`	`temp = (output - true)` `output = temp/len(true)`

Note that the delta computation for softmax is special because it's only used for the last layer. There's a bit more going on (theoretically) than we have time to discuss here. We'll discuss this more in a later chapter. For now, let's install some better activation functions in our MNIST classification network.

```python
import numpy as np
np.random.seed(1)

def tanh(x):
    return np.tanh(x)

def tanh2deriv(output):
    return 1 - (output ** 2)

def softmax(x):
    temp = np.exp(x)
    return temp / np.sum(temp, axis=1, keepdims=True)

alpha, iterations, hidden_size = (2, 300, 100)
pixels_per_image, num_labels = (784, 10)
batch_size = 100

weights_0_1 = 0.02*np.random.random((pixels_per_image,hidden_size))-0.01
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
        correct_cnt = 0
        for i in xrange(len(images) / batch_size):
            batch_start, batch_end=((i * batch_size),((i+1)*batch_size))
            layer_0 = images[batch_start:batch_end]
            layer_1 = tanh(np.dot(layer_0,weights_0_1))
            dropout_mask = np.random.randint(2,size=layer_1.shape)
            layer_1 *= dropout_mask * 2
            layer_2 = softmax(np.dot(layer_1,weights_1_2))

            for k in xrange(batch_size):
                correct_cnt += int(np.argmax(layer_2[k:k+1]) == \
                        np.argmax(labels[batch_start+k:batch_start+k+1]))

            layer_2_delta = (labels[batch_start:batch_end]-layer_2)\
                    / (batch_size * layer_2.shape[0])
            layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
                    * tanh2deriv(layer_1)
            layer_1_delta *= dropout_mask

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

        test_correct_cnt = 0

        for i in xrange(len(test_images)):

            layer_0 = test_images[i:i+1]
            layer_1 = relu(np.dot(layer_0,weights_0_1))
            layer_2 = np.dot(layer_1,weights_1_2)

            test_correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(test_labels[i:i+1]))
        if(j % 10 == 0):
            sys.stdout.write("\n"+ \
             "I:" + str(j) + \
             " Test-Acc:"+str(test_correct_cnt/float(len(test_images)))+\
             " Train-Acc:" + str(correct_cnt/float(len(images))))
```

Upgrading our MNIST Network

Let's upgrade our MNIST network to reflect what we'e learned.

Theoretically, our tanh function should make for a better hidden layer activation and our softmax should make for a better output layer activation function. When we test them, they do in fact reach a higher score. However, things are not always as simple as they seem.

I had to make a couple of adjustments in order to have the network "tuned" properly with these new activations. For tanh, I had to reduce the standard deviation of the incoming weights. Remember that we initialize our weights randomly. np.random.random simply creates a random matrix with numbers randomly spread between 0 and 1. By multiplying by 0.2 and subtracting by 0.1, we rescale this random range to be between -0.1 and 0.1. This worked great for relu, but is less optimal for tanh. Tanh likes to have a narrower random initialization, so I adjusted it to be between -0.01 and 0.01.

Furthermore, I removed our "error" calculation because we're not ready for that yet. Technically, softamx is best used with an error function called Cross Entropy. This network properly computes the layer_2_delta for this error measure, but since we haven't analyzed why this error function is advantageous, I removed the lines to compute it.

Finally, as with almost all changes you make to a neural network, I had to revisit our "alpha" tuning. I found that a much higher alpha was required to reach a good score within 300 iterations.

And wallah! As expected, we reached a higher testing accuracy of 82.6% accuracy!

```
I:0 Test-Acc:0.273 Train-Acc:0.144      I:150 Test-Acc:0.818 Train-Acc:0.935
I:10 Test-Acc:0.63 Train-Acc:0.732      I:160 Test-Acc:0.817 Train-Acc:0.937
I:20 Test-Acc:0.643 Train-Acc:0.736     I:170 Test-Acc:0.82 Train-Acc:0.936
I:30 Test-Acc:0.673 Train-Acc:0.78      I:180 Test-Acc:0.826 Train-Acc:0.947
I:40 Test-Acc:0.709 Train-Acc:0.811     I:190 Test-Acc:0.823 Train-Acc:0.941
I:50 Test-Acc:0.741 Train-Acc:0.839     I:200 Test-Acc:0.822 Train-Acc:0.948
I:60 Test-Acc:0.757 Train-Acc:0.859     I:210 Test-Acc:0.821 Train-Acc:0.955
I:70 Test-Acc:0.773 Train-Acc:0.867     I:220 Test-Acc:0.824 Train-Acc:0.951
I:80 Test-Acc:0.785 Train-Acc:0.884     I:230 Test-Acc:0.823 Train-Acc:0.947
I:90 Test-Acc:0.788 Train-Acc:0.901     I:240 Test-Acc:0.826 Train-Acc:0.958
I:100 Test-Acc:0.787 Train-Acc:0.894    I:250 Test-Acc:0.824 Train-Acc:0.956
I:110 Test-Acc:0.795 Train-Acc:0.909    I:260 Test-Acc:0.822 Train-Acc:0.959
I:120 Test-Acc:0.8 Train-Acc:0.916      I:270 Test-Acc:0.821 Train-Acc:0.965
I:130 Test-Acc:0.811 Train-Acc:0.918    I:280 Test-Acc:0.817 Train-Acc:0.963
I:140 Test-Acc:0.812 Train-Acc:0.923    I:290 Test-Acc:0.82 Train-Acc:0.961
```

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*

What does it mean to Understand Language?

What kinds of predictions do people do on language?

Up until now, we've been using neural networks to model image data. However, neural networks can be used to understand a much wider variety of dataset. Exploring new datasets also teaches us a lot about neural networks in general, as different datasets often justify different styles of neural network training according the challenges hidden in the data.

Machine Learning

Image Recognition

Deep Learning

Artificial Intelligence

Natural Language Processing
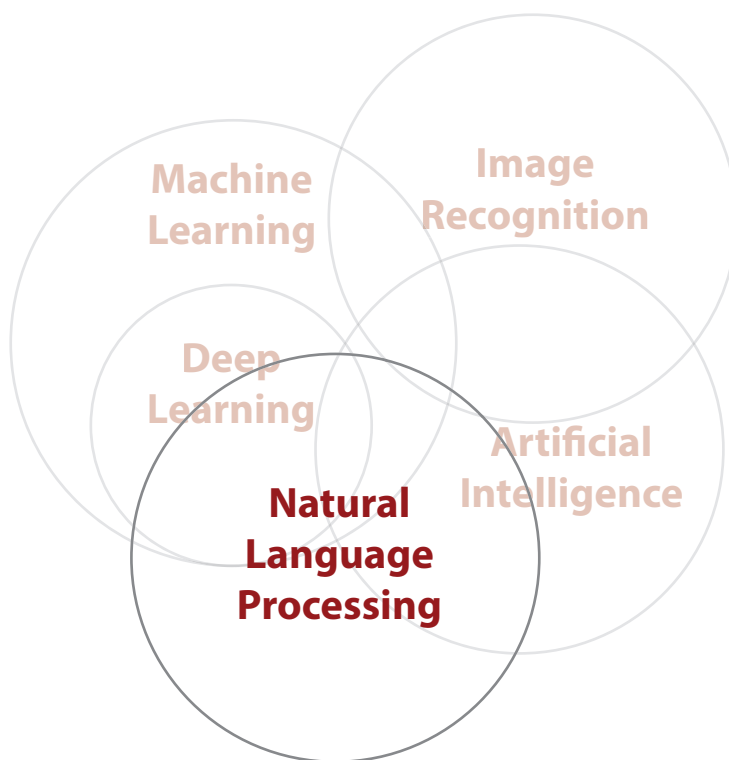
We begin by exploring a much older field that overlaps with Deep Learning called Natural Language Processing (NLP). This field is dedicated exclusively to the automated understanding of human language (previously not using Deep Learning). In this chapter, we're going to discuss the basics of Deep Learning's approach to this field.

Natural Language Processing (NLP)

Natural Langauge Processing is divided into a collection of tasks or challenges.

Perhaps the best way to quickly get to know Natural Language Processing is by considering a few of the many challenges that the NLP community seeks to solve. Here are a few types of classification problem that are common to NLP.

- Using the **characters** of a document to predict *where words start and end.*
- Using the **words** of a document to predict where sentences start and end.
- Using the **words in a sentence** to predict *the part of speech for each word*
- Using **words in a sentence** to predict *where phrases start and end.*
- Using **words in a sentence** to predict *where named entity (person, place, thing) references start and end..*
- Using **sentences in a document** to predict *which pronouns refer to the same person / place / thing.*
- Using **words in a sentence** to predict the *sentiment* of a sentence.
- and more...

Generally speaking, NLP tasks seek to do one of three things. A task is either labeling a region of text (such as Part-of-Speech Tagging, Sentiment Classification, or Named Entity Recognition), linking two or more regions of text (such as Coreference, which tries to answer whether two mentions of a "real world thing" are in fact referencing the same "real world thing" where "real world thing" is generally a person, place, or some other named entity), or trying to fill in missing information (missing words) based on context.

Perhaps it is also apparent how Machine Learning and Natural Language Processing are deeply intertwined. Until recently, most state-of-the-art NLP algorithms were advanced, probabilistic, non-parametric models (not Deep Learning). However, the recent development and popularization of two major neural algorithms have swept the field of NLP, namely neural word embeddings and recurrent neural networks.

In this chapter, we're going to build a word embedding algorithm and demonstrate why it increases the accuracy of NLP algorithms. In the next chapter, we're doing to create a recurrent neural network and demonstrate why it is so effective at predicting across sequences.

It is also worth mentioning the key role that NLP (perhaps leveraging Deep Learning) plays in the advancement of Artificial Intelligence. A.I. seeks to create machines that can think and engage with the world as humans do (and beyond). NLP plays a very special role in this endeavor as language is the bedrock of conscious logic and communication in humans. As such, the methods by which machines can leverage and understand language form the foundation of "human-like" logic in machines- the foundation of "thought".

Supervised NLP

Words go in and predictions come out.

Perhaps you will remember the picture below from Chapter 2. Supervised learning is all about taking "what we do know" and transforming it into "what we want to know." Up until now "what we know" has always comprised numbers in one way or another. However, NLP leverages text as input. How do we process this?

what we know → *supervised learning* → what we want to know

Well, since neural networks only map input numbers to output numbers, our first step is to convert our text into numerical form. Much like we converted our streetlight dataset before, we need to convert our real world data (in this case text) into a *matrix* that the neural network can consume. As it turns out, how we do this is extremely important!

raw text → matrix of # → *supervised learning* → what we want to know

How should we convert text to numbers? Answering that question requires some thought regarding our problem. Remember, neural networks look for correlation between their input and output layers. Thus, we want to convert our text into numbers in such a way that the correlation between input and output is *most obvious* to the network. This will make for faster training and better generalization.

In order to know what input format makes input/output correlation the most obvious to the network, we need to know what our input/output dataset looks like. So, to explore this topic, we're going to take on the challenge of Topic Classification.

IMDB Movie Reviews Dataset

Predicting whether people post positive or negative reviews.

The IMDB movie reviews dataset is a collection of review -> rating pairs that often look like the following (note: this is an imitation not actually pulled from IMDB).

```
"This movie was
terrible! The plot
was dry, the acting          Rating: 1 (stars)
unconvincing, and
I spilled popcorn
on my shirt.
```

The entire dataset comprises around 50,000 of these pairs, where the input reviews are usually a few sentences and the output ratings are between 1 and 5 stars. People consider it a "sentiment dataset" because the stars are very indicative of the overall sentiment of the movie review. However, it should be quite obvious that this "sentiment" dataset might be very different from other sentiment datasets, such as product reviews, or hospital patient reviews.

What we want to do is train a neural network that can leverage the input text to make accurate predictions of the output score. To accomplish this, we must first decide how to turn our input and output datasets into matrices. Interestingly, our output dataset is a number, which perhaps makes it an easier place to start. We'll adjust the range of stars to be between 0 and 1 instead of 1 and 5 so that we can use binary softmax. This is really all we need to do to the output. I'll show an example on the next page.

The input data, however, is a bit tricker. To begin, let's just consider the raw data itself. It's a list of characters. This presents a few problems, not only is the input data text instead of numbers, but it's *variable length* text. So far, our neural networks always take an input of a fixed size. We'll need to overcome this.

So, the raw input simply won't work. The next question to ask yourself is, "What about this data will have correlation with the output?" Simply representing that property might work quite well. For starters, I wouldn't expect any characters (in our list of characters) to have any correlation with the sentiment. We need to think about it differently.

What about the words? There are several words in this dataset that would have quite a bit of correlation! I would bet that "terrible" and "unconvincing" have significant *negative* correlation with the rating. By *negative*, I mean that as they *increase* in frequency in any input datapoint (any review), the tends to rating *decrease*.

Perhaps this property is more general! Perhaps words by themselves (even out of context) would have significant correlation with sentiment. Let's explore this further.

Capturing Word Correlation in Input Data

"Bag of Words": Given a review's vocabulary, predict the sentiment.

      If we observe that there is correlation between the vocabulary of an IMDB review and its rating, then we can proceed to the next step, creating an input matrix that represents the vocabulary of a movie review.

      What is commonly done in this case is to create a matrix where each row (vector) corresponds to each movie review, and each column represents whether a review contains a particular word in our vocabulary. So, to create the vector for a review, we calculate the vocabulary of the review and then put a "1" in each corresponding column for that review, and "0"s everywhere else. How big are these vectors? Well, if there are 2000 words, and we need to have a place in each vector for each word, then each vector will have 2000 dimensions. This form of storage is called "one-hot encoding" and is the most common format for encoding binary data (the "binary" presence or absence of an input datapoint amongst a vocabulary of possible input datapoints). So, if our vocabulary was only 4 words, our one-hot encoding might look like the following.

```
import numpy as np

onehots = {}
onehots['cat'] = np.array([1,0,0,0])
onehots['the'] = np.array([0,1,0,0])
onehots['dog'] = np.array([0,0,1,0])
onehots['sat'] = np.array([0,0,0,1])

sentence = ['the','cat','sat']
x = word2hot[sentence[0]] + \
    word2hot[sentence[1]] + \
    word2hot[sentence[2]]

print("Sent Encoding:" + str(encoding))
```

cat   (1)(0)(0)(0)

the   (0)(1)(0)(0)

dog   (0)(0)(1)(0)

sat   (0)(0)(0)(1)
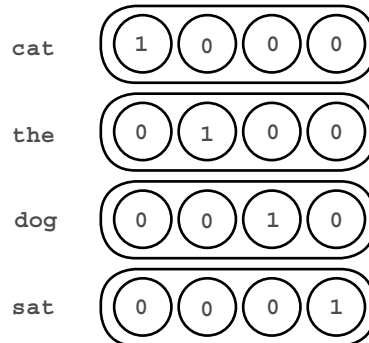
As you can see, we create a vector for each term in the vocabulary and this allows us to use simple vector addition to create a vector representing a subset of the total vocabulary (such as a subset corresponding to the words in a sentence).

"the cat sat"

(1)(1)(0)(1)

Code Output: Sent Encoding:[1 1 0 1]
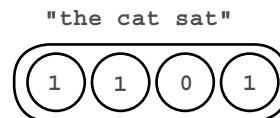
Note that when we create an embedding for several (such as "the cat sat"), we have multiple options in the case that words occur multiple times. If our phrase was "cat cat cat", we could either sum the vector for "cat" three times (resulting in [3,0,0,0]) or just take the unique "cat" a single time (resulting in [1,0,0,0]). The latter typically works better for language.

Predicting Movie Reviews

With our encoding strategy and our previous network, we can predict sentiment.

So, using the strategy identified on the previous page, we can build a vector for each word in our sentiment dataset and use our previous 2 layer network to predict sentiment. While I'm going to give you the code below, I strongly recommend attempting this from memory. Open up a new Jupyter notebook, load in the dataset, build your one-hot vectors, and then build a neural network to predict the rating of each movie review (positive or negative). Below is how I would do the pre-processing step.

```
import sys

f = open('reviews.txt')
raw_reviews = f.readlines()
f.close()

f = open('labels.txt')
raw_labels = f.readlines()
f.close()

tokens = map(lambda x:set(x.split(" ")),raw_reviews)

vocab = set()
for sent in tokens:
    for word in sent:
        if(len(word)>0):
            vocab.add(word)
vocab = list(vocab)

word2index = {}
for i,word in enumerate(vocab):
    word2index[word]=i

input_dataset = list()
for sent in tokens:
    sent_indices = list()
    for word in sent:
        try:
            sent_indices.append(word2index[word])
        except:
            ""
    input_dataset.append(list(set(sent_indices)))

target_dataset = list()
for label in raw_labels:
    if label == 'positive\n':
        target_dataset.append(1)
    else:
        target_dataset.append(0)
```

Intro to an Embedding Layer

One more trick to make our network faster

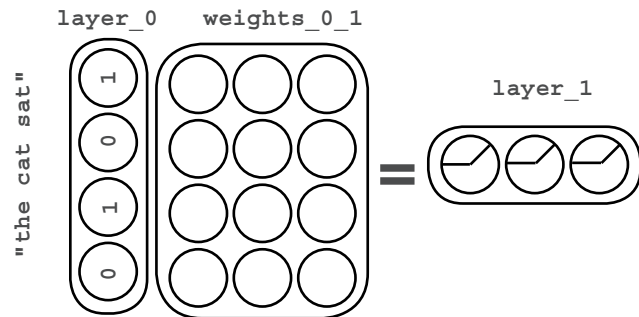layer_2

weights_1_2

layer_1

weights_0_1

layer_0

      On the left I've written the diagram from our previous neural network that we're now going to use to predict senti-ment. However, before we actually do the implementation, I want to describe the layer names here. Our first "layer" is our dataset (layer_0). This is followed by what's called a "Linear" layer (weights_0_1). This is followed by a Relu layer (layer_1), another Linear layer (weights_1_2) and then the output, which is our "prediction" layer. As it turns out, we can actually take a bit of a shortcut to layer_1 by replacing our first Linear layer (weights_0_1) with an Embedding layer.

      As it turns out, taking a vector of 1s and 0s is math-ematically equivalent to simply summing several rows of a matrix. Thus, it is actually much more efficient to simply select the relevant rows of **weights_0_1** and sum them as opposed to doing a big vector-matrix multiplication. Since our Sentiment vocabulary is on the order of 70,000 words, most of the vector-matrix multiplication is spent multiplying 0s in the input vector by different rows of the ma-trix before summing them. Simply selecting the rows corresponding to each word in a matrix and summing them is much more efficient.

      Using this process of select-ing rows and performing a sum (or average) means that we're treating our first Linear layer (weights_0_1) as an Embedding layer. Structural-ly, they are identical (layer_1 is the same identical using either method for forward propagation). The only difference is that summing a small number of rows is much faster
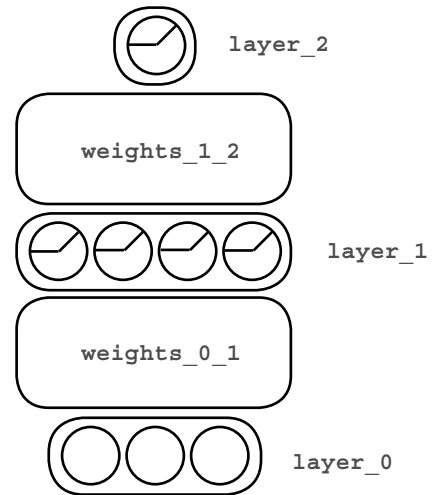
One-Hot Vector-Matrix Multplication

layer_0   weights_0_1   layer_1

"the cat sat"

Matrix Row Sum

weights_0_1   layer_1

the

+

sat

+

cat
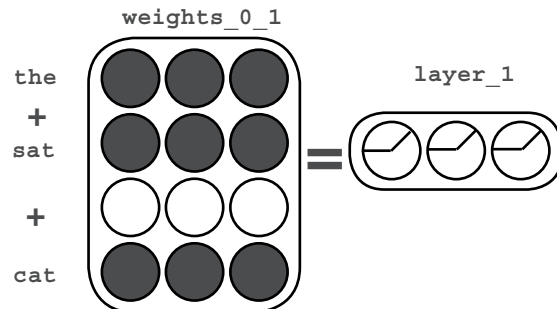
Predicting Movie Reviews

After running the code from two pages ago, run this code.

```python
import numpy as np
np.random.seed(1)

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

alpha, iterations = (0.01, 2)
hidden_size = 100

weights_0_1 = 0.2*np.random.random((len(vocab),hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,1)) - 0.1

correct,total = (0,0)
for iter in range(iterations):

    # train on first 24,000
    for i in range(len(input_dataset)-1000):

        x,y = (input_dataset[i],target_dataset[i])
        layer_1 = sigmoid(np.sum(weights_0_1[x],axis=0)) #embed + sigmoid
        layer_2 = sigmoid(np.dot(layer_1,weights_1_2)) # linear + softmax

        layer_2_delta = layer_2 - y # compare pred with truth
        layer_1_delta = layer_2_delta.dot(weights_1_2.T) #backprop

        weights_0_1[x] -= layer_1_delta * alpha
        weights_1_2 -= np.outer(layer_1,layer_2_delta) * alpha

        if(np.abs(layer_2_delta) < 0.5):
            correct += 1
        total += 1
        if(i % 10 == 9):
            progress = str(i/float(len(input_dataset)))
            sys.stdout.write('\rIter:'+str(iter)\
                             +' Progress:'+progress[2:4]\
                             +'.'+progress[4:6]\
                             +'% Training Accuracy:'\
                             + str(correct/float(total)) + '%')
    print
correct,total = (0,0)
for i in range(len(input_dataset)-1000,len(input_dataset)):

    x = input_dataset[i]
    y = target_dataset[i]

    layer_1 = sigmoid(np.sum(weights_0_1[x],axis=0))
    layer_2 = sigmoid(np.dot(layer_1,weights_1_2))

    if(np.abs(layer_2 - y) < 0.5):
        correct += 1
    total += 1
print("Test Accuracy:" + str(correct / float(total)))
```

Interpreting the Output

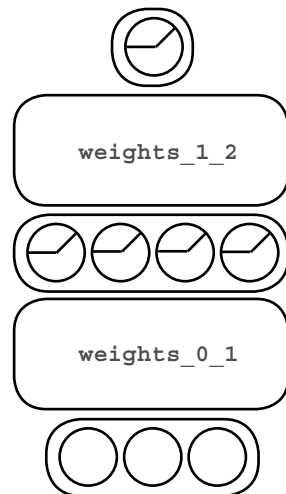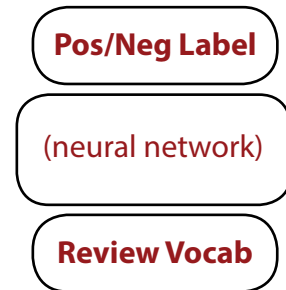### What did our neural network learn along the way?

Below, you can see the output of our Movie Reviews neural network. From one perspective, this is simply the same Correlation Summarization that we have already discussed.

```
Iter:0 Progress:95.99% Training Accuracy:0.831083333333%
Iter:1 Progress:95.99% Training Accuracy:0.865770833333%
Test Accuracy:0.853
```

From this perspective, the neural network on the previous page was simply looking for correlation between the input datapoints and the output datapoints. However, those datapoints have characteristics that we are quite familiar with (notably those of language). Furthermore, it is extremely beneficial to consider what patterns of language would be detected by the Correlation Summarization, and more importantly, which ones would not. After all, just because the network is able to find correlation between our input and output datasets does not mean that it understands every useful pattern of language. Furthermore, understanding the difference between what the network (in its current configuration) is capable of learning realtive to what it needs to know to properly understand language is an incredibly fruitful line of thinking. This is what researchers on the front lines of state-of-the-art research consider, and it's what we're going to consider today.

So, what about language did our movie reviews network actually learn? Well, let's first start by considering what we presented to the network. As displayed in the diagram on the top right, we presented each review's vocabulary as input, and asked it to predict one of two labels (positive or negative). So, given that our Correlation Summarization says the network will look for correlation between our input and output datasets, at a very minimum, we would expect our network to identify words that have either a positive or negative correlation (by themselves).

This follows naturally from the Correlation Summarization itself. We present the presence or absense of a word. As such, the Correlation Summarization will find direct correlation between this presence/absence and each of our two labels. However, this isn't the whole story.

Neural Architecture

How did our choice of architecture affect what the network learned?

On the last page, we discussed the first, most trivial type of information that our neural network learned: direct correlation between our input and target datasets. This observation is largely the "clean slate" of neural intelligence. (If your network cannot discover direct correlation between input and output data, something is probably broken). The development of more sophisticated architectures is based on the need to find more complex patterns than direct correlation, and this network is no exception.

The minimial architecture needed to identify direct correlation is a 2 layer network, where a network has a single weight matrix which connects directly from the input layer to the output layer. However, we used a network that has a hidden layer. This begs the question, what does this hidden layer do?

Fundamentally, hidden layers are about grouping datapoints from a previous layer into "n" groups (where "n" is the number of neurons in the hidden layer). Each hidden neuron takes in a datapoint and answers the question "is this datapoint in my group?". As the hidden layer learns, it searches for useful groupings of its input. What are useful groupings?

An input datapoint "grouping" is useful if it does two things. First and foremost, the grouping must be useful to the prediction of an output label. If it's not useful to the output prediction, <u>the Correlation Summarization will never lead the network to find the group</u>. This is a hugely valuable realization. Much of neual network research is about finding training data (or some other manufactured "signal" for the network to artificially predict) so that it finds groupings that are useful for a task (such as predicting movie review stars). We'll discuss this more in a moment.

Secondly, a "grouping" is useful if it is an actual phenomenon in the data that we care about. Bad groupings just memorize the data. Good groupings pick up on phenomena that are useful linguistically. For example, when predicting whether a movie review is positive or negative, understanding the difference between "terrible" and "NOT terrible" is a powerful grouping. We would love to have a neuron that turned OFF when it saw "<polarized adjective>" and turned ON when it saw "not <polarized adjective>". This would be a powerful grouping for the next layer to use to make the final prediction. However, as the input to our neural network is simply the vocabulary of a review, "it was great, not terrible" creates exactly the same "layer_1" value as "it was terrible, not great". For this reason, we know that our network is very unlikely to create a hidden neuron that understands negation. In fact, this means of testing whether a layer is the same or different based on a certain language pattern is a great first step for knowing whether an architecture is likely to find said pattern using the correlation summarization. If you can construct two examples with an identical hidden layer, one with the pattern you find interesting and one without, the network is unlikely to find that pattern.

Neural Architecture (cont.)

How did our choice of architecture affect what the network learned?

As we learned on the last page, a hidden layer fundamentally groups the previous layer's data. At a granular level, each neuron classifies a datapoint as either subscrbing or not subscribing to its group. At a higher level, two datapoints (i.e., "movie reviews") are similar if they subscribe to many of the same groups. Finally, two inputs (i.e., words) are similar if the weights linking them to various hidden neurons (a measure of each word's "group affinity") are similar. So, given this knowledge, in our previous neural network, what should we observe in the weights going into the hidden neurons from the words?

What should we see in the weights connecting words and hidden neurons?

Here's a hint. Words that have a similar predictive power should subscribe to similar groups (hidden neuron configurations). So, what does this mean for the weights connecting each word to each hidden neuron?

Here's the answer. Words that correlate with similar labels (positive or negative) will have similar weights connecting them to various hidden neurons. This is because the neural network learns to bucket them into similar hidden neurons so that the final layer (weights_1_2) can make the correct positive or negative predictions. We can see this phenomenon by taking a particularly positive or negative word and searching for the other words with the most similar weight values. In other words, we can take each word and see which other words have the most similar "weight values" connecting them to each hidden neuron (to each "group"). Words that subscribe to similar groups will have similar predictive power for positive or negative labels. As such, words that subscribe to similar groups, having similar weight values, will also have similar meaning. Abstractly, in terms of neural networks, a neuron has similar meaning to other neurons in the same layer if and only if it has similar weights connecting it to the next and/or previous layers.
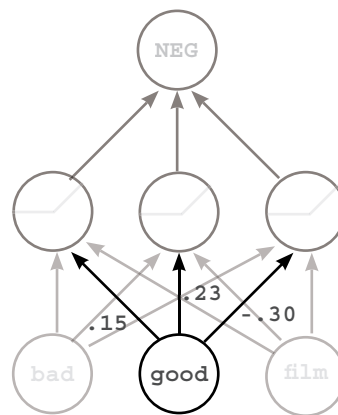
The 3 bold weights
for "good" form the
"embedding" for good.
They reflect how much
the term "good" is a
member of each group
(i.e. hidden neuron)
Words with similar
predictive power will
have similar word
embeddings (weight
values)

Comparing Word Embeddings

How can we visualize weight similarity?

For each input word, we can select the list of weights proceeding out of it to the various hidden neurons by simply selecting the corresponding row of weights_0_1. Each entry in the row represents each weight proceeding from that row's word to each hidden neuron. Thus, to figure out which words are most similar to a target term, we simply compare each word's vector (row of the matrix) to that of the target term. Our comparison of choice is called "Euclidian Distance" which we can perform as seen in the code below.

```
from collections import Counter
import math

def similar(target='beautiful'):
    target_index = word2index[target]
    scores = Counter()
    for word,index in word2index.items():
        raw_difference = weights_0_1[index] - (weights_0_1[target_index])
        squared_difference = raw_difference * raw_difference
        scores[word] = -math.sqrt(sum(squared_difference))

    return scores.most_common(10)
```

This allows us to easily query for the most similar word (neuron) according to the network.

```
print(similar('beautiful'))                print(similar('terrible'))

[('beautiful', -0.0),                      [('terrible', -0.0),
 ('atmosphere', -0.70542101298),            ('dull', -0.760788602671491),
 ('heart', -0.7339429768542354),            ('lacks', -0.76706470275372),
 ('tight', -0.7470388145765346),            ('boring', -0.7682894961694),
 ('fascinating', -0.7549291974),            ('disappointing', -0.768657),
 ('expecting', -0.759886970744),            ('annoying', -0.78786389931),
 ('beautifully', -0.7603669338),            ('poor', -0.825784172378292),
 ('awesome', -0.76647368382398),            ('horrible', -0.83154121717),
 ('masterpiece', -0.7708280057),            ('laughable', -0.8340279599),
 ('outstanding', -0.7740642167)]            ('badly', -0.84165373783678)]
```

As you might expect, the most similar term to every word is simply itself, followed by words which had similar usefulness as the target term. Again, as you might expect, since the network really only has two labels (positive/negative), the input terms are grouped according to which label they tend to predict. This is a standard phenomenon of the correlation summarization. It seeks to create similar representations (layer_1 values) within the network based on the label being predicted so that it can predict the right label. In this case, the side effect is that the weights feeding into layer_1 get grouped according to output label. The key takeaway is a gut instinct on this phenomenon of the correlation summarization. It consistently attempts to convince the hidden layers to be similar based on which label should be predicted.

What is the Meaning of a Neuron?

Meaning is entirely based on the target labels being predicted

Note that the meanings of different words didn't totally reflect how we might group them. Notice that the most similar term to "beautiful" is "atmosphere". This is actually a very valuable lesson. For the purposes of predicting whether or not a movie review is positive or negative, these words have nearly identical meaning. However, in the real world, their meaning is quite different (one is an adjective and another a noun, for example).

```
print(similar('beautiful'))              print(similar('terrible'))

[('beautiful', -0.0),                    [('terrible', -0.0),
 ('atmosphere', -0.70542101298),          ('dull', -0.760788602671491),
 ('heart', -0.7339429768542354),          ('lacks', -0.76706470275372),
 ('tight', -0.7470388145765346),          ('boring', -0.7682894961694),
 ('fascinating', -0.7549291974),          ('disappointing', -0.768657),
 ('expecting', -0.759886970744),          ('annoying', -0.78786389931),
 ('beautifully', -0.7603669338),          ('poor', -0.825784172378292),
 ('awesome', -0.76647368382398),          ('horrible', -0.83154121717),
 ('masterpiece', -0.7708280057),          ('laughable', -0.8340279599),
 ('outstanding', -0.7740642167)]          ('badly', -0.84165373783678)]
```

This realization is incredibly important. "Meaning" (of a neuron) in our network is defined based on our target labels. Everything in the neural network is contexualized based on the correlation summarization trying to correctly make predictions. Thus, even though you and I know a great deal about these words, the neural network is entirely ignorant of all information outside of the task at hand. So, how can we convince our network to learn more nuanced information about neurons (in this case, word neurons)? Well, if we give it input and target data that requires a more nuanced understanding of language, it will have reason to learn more nuanced interpretations of various terms. So, what should we use our neural network to predict so that it learns more interesting weight values for our word neurons?

Mary had a little lamb whose _____**????**_____ was white as snow .

The task that we're going to use to learn more interesting weight values for our word neurons is a glorified "fill in the blank" task. Why are we going to use this? Well, first off, we have nearly infinite training data (the internet), which means we have nearly infinite "signal" for the neural network to use to learn more nuanced information about words. Furthermore, being able to accurately fill in the blank requires at least some notion of context about the real world. For example, in the blank above, is it more likely that the blank is correcty filled by the word "anvil" or "wool"? Let's see if our neural network can figure it out.

Filling in The Blank

Learning richer meanings for words by having a richer signal to learn.

      In this next neural network, we're going to use almost exactly the same network as the previous example with only a few modifications. First, instead of predicting a single label given a movie review, we're going to take each (5 word) phrase, remove one word (a focus term), and attempt to train a network to figure out the identity of the word we removed using the rest of the phrase. Secondly, we're going to use a trick called "Negative Sampling" to make our network train a bit faster. Consider that in order to predict which term is missing, we have to have one label for each possible word. This means that we have several thousand labels, which would cause our network to train quite slowly. To overcome this, we're going to randomly ignore most of our labels for each forward propagation step (as in, pretend they don't exist at all). While this might seem like a crude approximation, it's a technique that works quite well in practice. Below is our pre-processing code for this example.

```python
import sys,random,math
from collections import Counter
import numpy as np

np.random.seed(1)
random.seed(1)
f = open('reviews.txt')
raw_reviews = f.readlines()
f.close()

tokens = list(map(lambda x:(x.split(" ")),raw_reviews))
wordcnt = Counter()
for sent in tokens:
    for word in sent:
        wordcnt[word] -= 1
vocab = list(set(map(lambda x:x[0],wordcnt.most_common())))

word2index = {}
for i,word in enumerate(vocab):
    word2index[word]=i

concatenated = list()
input_dataset = list()
for sent in tokens:
    sent_indices = list()
    for word in sent:
        try:
            sent_indices.append(word2index[word])
            concatenated.append(word2index[word])
        except:
            ""
    input_dataset.append(sent_indices)
concatenated = np.array(concatenated)
random.shuffle(input_dataset)
```

Filling in The Blank (cont)

Learning richer meanings for words by having a richer signal to learn.

```python
alpha, iterations = (0.05, 2)
hidden_size,window,negative = (50,2,5)

weights_0_1 = (np.random.rand(len(vocab),hidden_size) - 0.5) * 0.2
weights_1_2 = np.random.rand(len(vocab),hidden_size)*0

layer_2_target = np.zeros(negative+1)
layer_2_target[0] = 1

def similar(target='beautiful'):
  target_index = word2index[target]

  scores = Counter()
  for word,index in word2index.items():
    raw_difference = weights_0_1[index] - (weights_0_1[target_index])
    squared_difference = raw_difference * raw_difference
    scores[word] = -math.sqrt(sum(squared_difference))
  return scores.most_common(10)

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

for rev_i,review in enumerate(input_dataset * iterations):
  for target_i in range(len(review)):

    # since it's really expensive to predict every vocabulary
    # we're only going to predict a random subset
    target_samples = [review[target_i]]+list(concatenated\
    [(np.random.rand(negative)*len(concatenated)).astype('int').tolist()])

    left_context = review[max(0,target_i-window):target_i]
    right_context = review[target_i+1:min(len(review),target_i+window)]

    layer_1 = np.mean(weights_0_1[left_context+right_context],axis=0)
    layer_2 = sigmoid(layer_1.dot(weights_1_2[target_samples].T))
    layer_2_delta = layer_2 - layer_2_target
    layer_1_delta = layer_2_delta.dot(weights_1_2[target_samples])

    weights_0_1[left_context+right_context] -= layer_1_delta * alpha
    weights_1_2[target_samples] -= np.outer(layer_2_delta,layer_1)*alpha

  if(rev_i % 250 == 0):
    sys.stdout.write('\rProgress:'+str(rev_i/float(len(input_dataset)
        *iterations)) + "   " + str(similar('terrible')))
  sys.stdout.write('\rProgress:'+str(rev_i/float(len(input_dataset)
        *iterations)))
```

```
Progress:0.99998 [('terrible', -0.0), ('horrible', -2.846300248788519),
('brilliant', -3.039932544396419), ('pathetic', -3.4868595532695967),
('superb', -3.6092947961276645), ('phenomenal', -3.660172529098085),
('masterful', -3.6856112636664564), ('marvelous', -3.9306620801551664),
```

Meaning is Derived from Loss

Learning richer meanings for words by having a richer signal to learn.

      With this new neural network, we can subjectively see that our word embeddings cluster in a rather different way. Whereas before it clustered words according to their likelihood to predict a Positive or Negative label, now they cluster based on their likelihood to occur within the same phrase (sometimes regardless of sentiment). See for yourself!

<table>
<tr><th>Predicting POS/NEG</th><th>Fill In The Blank</th></tr>
</table>

```
print(similar('terrible'))              print(similar('terrible'))

[('terrible', -0.0),                    [('terrible', -0.0),
 ('dull', -0.760788602671491),           ('horrible', -2.79600898781),
 ('lacks', -0.76706470275372),           ('brilliant', -3.3336178881),
 ('boring', -0.7682894961694),           ('pathetic', -3.49393193646),
 ('disappointing', -0.768657),           ('phenomenal', -3.773268963),
 ('annoying', -0.78786389931),           ('masterful', -3.8376122586),
 ('poor', -0.825784172378292),           ('superb', -3.9043150978490),
 ('horrible', -0.83154121717),           ('bad', -3.9141673639585237),
 ('laughable', -0.8340279599),           ('marvelous', -4.0470804427),
 ('badly', -0.84165373783678)]           ('dire', -4.178749691835959)]


print(similar('beautiful'))             print(similar('beautiful'))

[('beautiful', -0.0),                    [('beautiful', -0.0),
 ('atmosphere', -0.70542101298),         ('lovely', -3.0145597243116),
 ('heart', -0.7339429768542354),         ('creepy', -3.1975363066322),
 ('tight', -0.7470388145765346),         ('fantastic', -3.2551041418),
 ('fascinating', -0.7549291974),         ('glamorous', -3.3050812101),
 ('expecting', -0.759886970744),         ('spooky', -3.4881261617587),
 ('beautifully', -0.7603669338),         ('cute', -3.592955888181448),
 ('awesome', -0.76647368382398),         ('nightmarish', -3.60063813),
 ('masterpiece', -0.7708280057),         ('heartwarming', -3.6348147),
 ('outstanding', -0.7740642167)]         ('phenomenal', -3.645669007)]
```

      The key takeaway here is that, even though we were training over the same dataset with a very similar architecture (3 layers, cross entropy, sigmoid nonlin), we can influence what the network learns within its weights by changing what we tell the network to predict. Thus, even though it's "looking at" the same statistical information, we can target what it learns based on what we select as the input and target values. For the moment, let's call this process of choosing what we want our network to learn "intelligence targeting".

      Controlling the input/target values is not the only way we can perform "intelligence targeting". We can also adjust how our network measures error, the size and types of layers that it has, and the types of regularization we apply. In Deep Learning research, all of these techniques fall under the underbrella of constructing what's called a **loss function.**

Meaning is Derived from Loss (cont.)

Neural networks don't really learn data... they minimize the loss function

In Chapter 4, we learned that learning is really about adjusting each weight in our neural network to bring our error down to 0. On this page, we're going to explain the exact same phenomena from a different perspective, choosing our error so that our neural network learns the patterns we're intersted in. Remember these lessons from Chapter 4?

The Golden Method for Learning

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

The Secret

For any `input` and `goal_pred`, there is an *exact relationship* defined between our `error` and **weight**, found by combining our **prediction** and **error** formulas.

$$\text{error} = ((0.5 * \text{weight}) - 0.8) ** 2$$

Perhaps you remember the formula above from our 1 weight neural network. In that network, we could evaluate our error by first forward propagating (0.5 * weight) and then comparing to our target (0.8). I would encourage you to think about this not from the perspective of two different steps (forward propagation, then error evaluation) but to instead consider the entire formula (including forward prop) to be the evaluation of an error value. This context will reveal the true cause of our different word embedding clusterings. Even though the network and datasets were similar, the error function was fundamentally different, leading to different word clusterings within each network.

Predicting POS/NEG

```
print(similar('terrible'))

[('terrible', -0.0),
 ('dull', -0.760788602671491),
 ('lacks', -0.76706470275372),
 ('boring', -0.7682894961694),
 ('disappointing', -0.768657),
 ('annoying', -0.78786389931),
 ('poor', -0.825784172378292),
 ('horrible', -0.83154121717),
 ('laughable', -0.8340279599),
```

Fill In The Blank

```
print(similar('terrible'))

[('terrible', -0.0),
 ('horrible', -2.79600898781),
 ('brilliant', -3.3336178881),
 ('pathetic', -3.49393193646),
 ('phenomenal', -3.773268963),
 ('masterful', -3.8376122586),
 ('superb', -3.9043150978490),
 ('bad', -3.9141673639585237),
 ('marvelous', -4.0470804427),
```

Meaning is Derived from Loss (cont.)

Our choice of loss function determines our neural network's knowledge.

The more formal term for an **error function** is a **loss function** or **objective function** (all three phrases are interchangable)**.** Considering learning to be all about minimizing a loss function (which includes our forward propagation) gives us a far broader perspective on how neural networks learn. We can have two neural networks with identical starting weights, trained over identical datasets that ultimately learn very different patterns because we choose a different loss function. In the case of our two movie review neural networks, our loss function was different because we chose two different target values (POS/NEG vs Fill In the Blank).

Different kinds of architectures, layers, regularization techniques, datasets, and non-linearities aren't really that different. These are simply the different ways we can choose to construct a loss function. Furthermore, if our network isn't learning properly, the solution can often come from any of these possible categories.

For example, if your network is overfitting, you can augment your loss function by choosing simpler nonlinearities, smaller layer sizes, shallower architectures, larger datasets, or more aggressive regularization techniques. All of these choices will have a fundamentally similar affect on your loss function and a similar consequence on the behavior of your network. They all interplay together, and over time you will learn how changing one can affect the performance of another, but for now, the important takeaway is that learning is about constructing a loss function and then minimizing it.

So, whenever you want a neural network to learn a pattern, everything you need to know to do so will be contained in your loss function. When we only had a single weight, this allowed our loss function to be quite simple, as you remember:

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

However, as you chain large numbers of complex layers together, your loss function will become more complicated (and that's ok). Just remember, if something is going wrong, the solution is in your loss function, which includes both your forward prediction and your raw error evaluation (such as mean squared error or cross entropy).

King - Man + Woman ~= Queen

Word analogies are an interesting consequence of the previously built network.

Before closing out this chapter, we should discuss what is, at the time of writing, still one of the most famous properties of neural word embeddings (word vectors like those we just created). The task of "filling in the blank" creates word embeddings with some interesting phenomena known as "Word Analogies", wherein you can take the vectors for different words and perform basic algebraic operations on them. For example, if you train the previous network on a large enough corpus, you'll be able to do the following: take the vector for "king", subtract from it the vector for "man" add in the vector for "woman" and then search for the most similar vector (other than those in your query). As it turns out, the most similar vector will often be the word "queen". We can even see similar phenomena in the "Fill In the Blank" network trained over movie reviews.

```
def analogy(positive=['terrible','good'],negative=['bad']):

    norms = np.sum(weights_0_1 * weights_0_1,axis=1)
    norms.resize(norms.shape[0],1)

    normed_weights = weights_0_1 * norms

    query_vect = np.zeros(len(weights_0_1[0]))
    for word in positive:
        query_vect += normed_weights[word2index[word]]
    for word in negative:
        query_vect -= normed_weights[word2index[word]]

    scores = Counter()
    for word,index in word2index.items():
        raw_difference = weights_0_1[index] - query_vect
        squared_difference = raw_difference * raw_difference
        scores[word] = -math.sqrt(sum(squared_difference))

    return scores.most_common(10)[1:]
```

terrible - bad + good ~=	**elizabeth - she + he ~=**
analogy(['terrible','good'],['bad'])	analogy(['elizabeth','he'],['she'])
[('superb', -223.3926217861),	[('christopher', -192.7003),
('terrific', -223.690648739),	('it', -193.3250398279812),
('decent', -223.7045545791),	('him', -193.459063887477),
('fine', -223.9233021831882),	('this', -193.59240614759),
('worth', -224.03031703075),	('william', -193.63049856),
('perfect', -224.125194533),	('mr', -193.6426152274126),
('brilliant', -224.2138041),	('bruce', -193.6689279548),
('nice', -224.244182032763),	('fred', -193.69940566948),
('great', -224.29115420564)]	('there', -193.7189421836)]

Word Analogies

**Linear compression of an existing property in the data.**

When this property was first discovered, it created a rather large flurry of exitement as people extrapolated many possible applications of such a technology. It is a rather amazing property in its own right, and it did create a veritable cottage industry around generating word embeddings of one variety or another. However, the word analogy property in and of itself hasn't grown that much since then, and most of the current work in language focuses instead on recurrent architectures (which we'll get to in a later chapter).

That being said, getting a good intuition for what's going on with word embeddings as a result of your chosen loss function is extremely valuable. We've already learned that our choice of loss function can affect how words are grouped, but this word analogy phenomenon is something different. What about our new loss function causes it to happen?

If you considered a word embedding having 2 dimensions, it's perhaps easier to envision exactly what it means for these word analogies to work.

```
king  = [0.6 , 0.1]
man   = [0.5 , 0.0]
woman = [0.0 , 0.8]
queen = [0.1 , 1.0]

king - man = [0.1 , 0.1]
queen - woman  = [0.1 , 0.2]
```
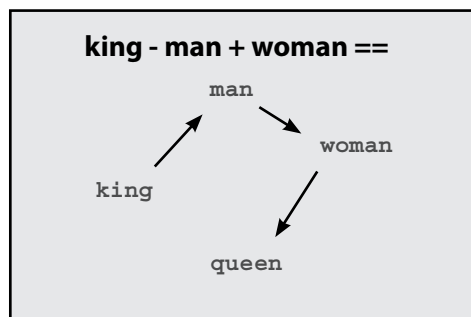

What this means it that the relative usefulness to the final prediction between king/man and queen/woman is similar. Why? The difference between king and man leaves a vector of "royalty". So, there's a bunch of male and female related words in one grouping, and then there's another grouping of "royal words" that's grouped in the "royal" direction.

This can be traced back to the loss that we chose. When the word "king" shows up in a phrase, it changes the probability of other words showing up in a certain way. It increases the probability of words related to "man" and the probability of words related to "royalty". "queen" showing up in a phrase increases the probability of words related to "woman" and the probability of words related to "royalty" (as a group). Thus, because they have this sort of ven-diagram of impact on the output probability, they end up subscribing to similar combinations of groupings. Oversimplifed, subscribes to the "male" and the "royal" dimensions of the hidden layer, while "queen" subscribes to the "female" and "royal" dimensions of the hidden layer. Thus, when you take the vector for "king" and subtract out some approximation of the "male" dimensions and add in the "female" ones, you get something close to "queen". The most important takeaway is that this is more about the properties of language than Deep Learning. Any linear compression of these co-occurrence statistics will behave similarly.

Conclusion

Neural word embeddings and the impact of loss on learning.

In this chapter, we've unpacked the fundamental principles of using neural networks to study language. We started with an overview of the primary problems in Natural Language Processing, proceeding to learn how neural networks model language at the word level using word embeddings. We further learned how our choice of loss function can change the kinds of properties that are captured by our word embeddings, and we finished with a discussion of perhaps the most magical of neural pheomena in this space, word analogies.

As with the other chapters, I would like to encourage you to build the examples in this chapter from scratch. While it might seem like this chapter is self standing, the lessons in loss function creation and tuning are invaluable and will be extremely important as we tackle increasingly more complicated strategies in future chapters. Good luck!