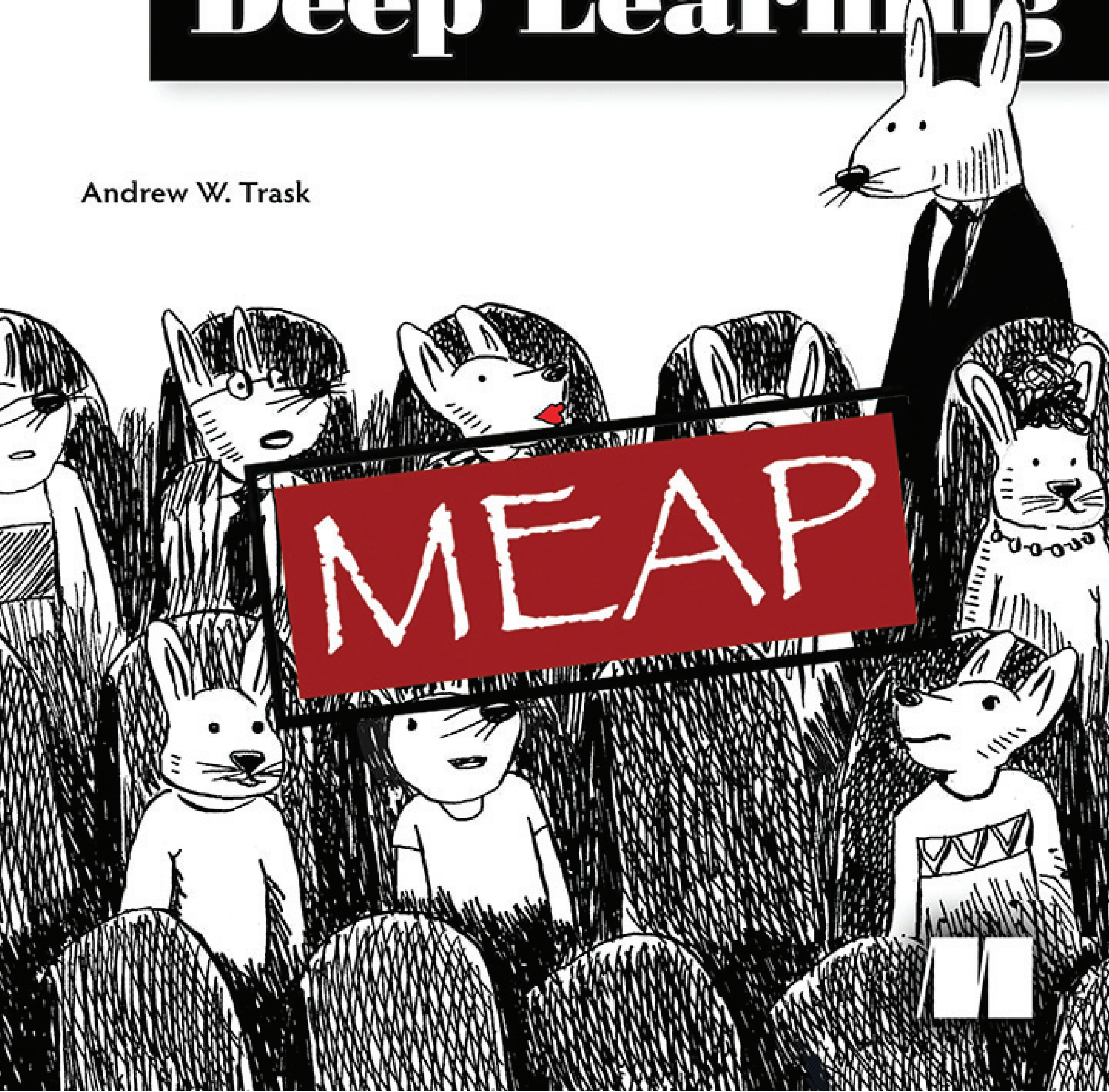
grokking

# Deep Learning

Andrew W. Trask



MEAP Edition
Manning Early Access Program
# Grokking Deep Learning
Version 5



For more information on this and other Manning titles go to
www.manning.com

# welcome

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

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

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

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

It is an honor to have your time and attention.

—Andrew Trask

# brief contents

**PART 1: NEURAL NETWORK BASICS**

1 *Introducing Deep Learning*

2 *Fundamental Concepts*

3 *Introduction to Neural Prediction: Forward Propagation*

4 *Introduction to Neural Learning: Gradient Descent*

5 *Learning Multiple Weights at a Time: Generalizing Gradient Descent*

6 *Building Your First "Deep" Neural Network: Introduction to Backpropagation*

7 *How to Picture Neural Networks: In Your Head and on Paper*

8 *Learning Signal and Ignoring Noise: Introduction to Regularization & Batching*

9 *Modeling Probabilities and Non-Linearities: Sigmoid, Tanh, Softmax*

**PART 2: ADVANCED LAYERS AND ARCHITECTURES**

10 *Neural Networks that Understand Edges and Corners: Convolutional Layers - Modeling Order*

11 *Neural Network Word Math (King - Man + Woman = Queen): Representation Layers - Modeling Sparsity*

12 *Writing Like Shakespeare + Translating English to Spanish: LSTMs - Modeling Sequences*

13 *Neural Networks that Read & Answer Questions: Modeling Memory and Attention*

14 *Building a Neural Network that Destroys You in Pong: Policy Gradient Reinforcement Learning - Modeling Environments & Reward*

15 *Where to go from here*

**APPENDIXES:**

A *Continuing Education*

B *Papers*

C *Frameworks (Torch, TensorFlow, Keras)*

# Introducing Deep Learning: Why you should learn it

# 1

## IN THIS CHAPTER

Why you should learn deep learning

Why you should read this book

What you need to get started

# Welcome to Grokking Deep Learning

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

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

# Why should you learn Deep Learning?

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

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

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

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

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

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

# Why you should read this book!

### Uniquely low barrier to entry

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

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

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

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

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

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

**Everything after the introduction chapters is "project" based.**

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

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

# What you need to get started

### Install Jupyter Notebook and the Numpy python library

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

### Pass High School Mathematics

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

### Find a personal problem you are interested in

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

# You'll probably need some Python knowledge

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

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

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

# Conclusion and Primer for Chapter 2

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

# Fundamental Concepts 2

## How do machines learn?

## IN THIS CHAPTER


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

What is a Parametric Model?

What is a Non-Parametric Model?

What is Supervised Learning?

What is Unsupervised Learning?

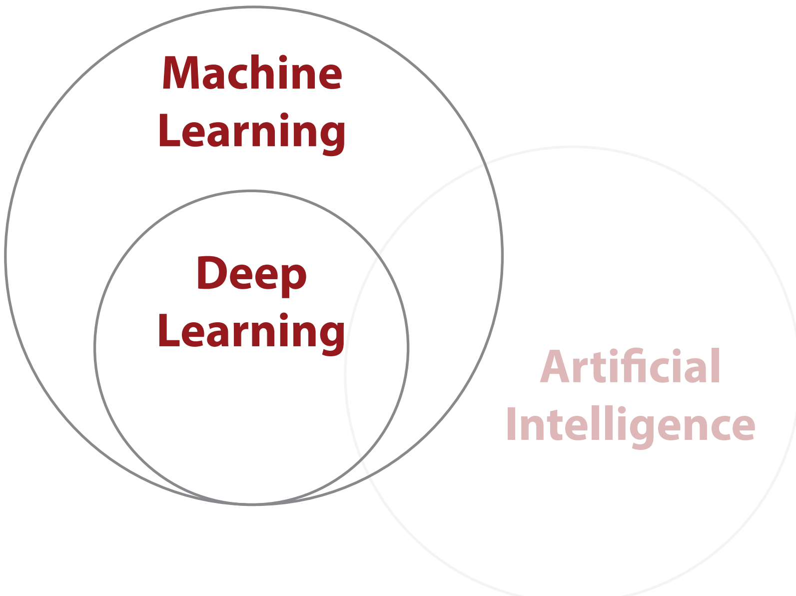
How can machines learn?


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

# What is Deep Learning?

### Deep Learning is a subfield of methods for Machine Learning

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




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

# What is Machine Learning?

> "A field of study that gives computers the ability to learn without being explicitly programmed'"
>
> \- Arthur Samuel

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

machine learning ~= monkey see monkey do

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

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

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

# Supervised Machine Learning

**Supervised Learning transforms one dataset into another.**

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


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

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

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


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

# Unsupervised Machine Learning

**Unsupervised Learning groups your data.**

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

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

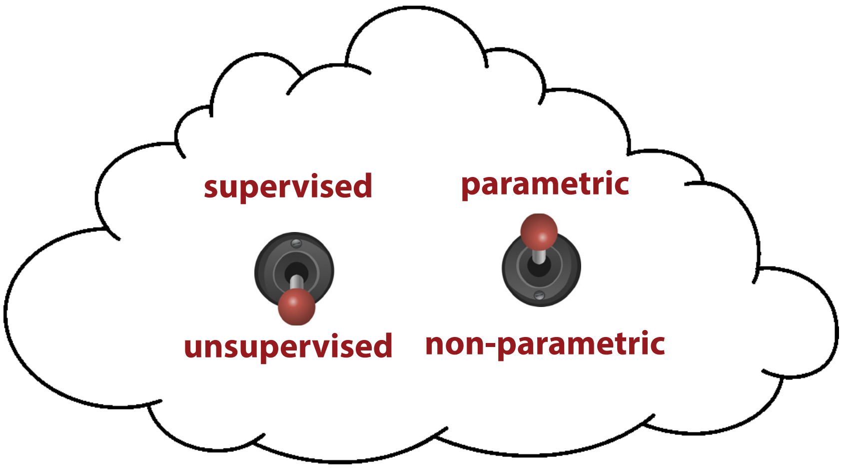

I have good news! This idea of clustering is something you can reliably hold onto in your mind as the definition of unsupervised learning. Even though there are many forms of unsupervised learning, *all forms of unsupervised learning can be viewed as a form of clustering*.


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

# Parametric vs Non-Parametric Learning

**Oversimplified: Trial and error learning versus counting and probability**

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




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

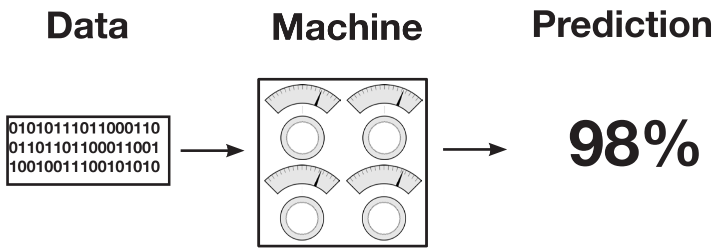
A parametric model is characterized by having a fixed number of parameters whereas a non-parametric model's number of parameters is infinite (determined by data).

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

# Supervised Parametric Learning

### Oversimplified: Trial and error learning using knobs

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




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

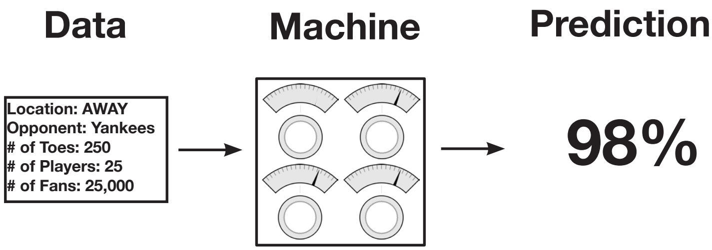
Perhaps it would "turn up" the "win/loss record" knob if the team's win/loss record was a good predictor. Inversely, it might turn down the "average number of toes" knob if that datapoint wasn't a good predictor. This is how parametric models learn!

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

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

## Step 1: Predict

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




## **Step 2: Compare** to Truth Pattern

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

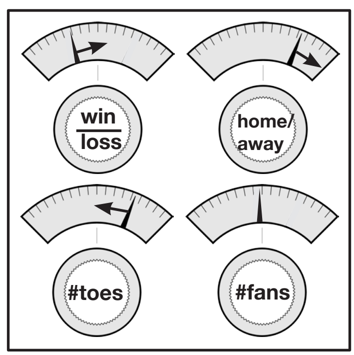
**Pred:** **98%** > Truth: **0%**

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

## **Step 3: Learn** the Pattern

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

**Adjusting Sensitivity By Turning Knobs**



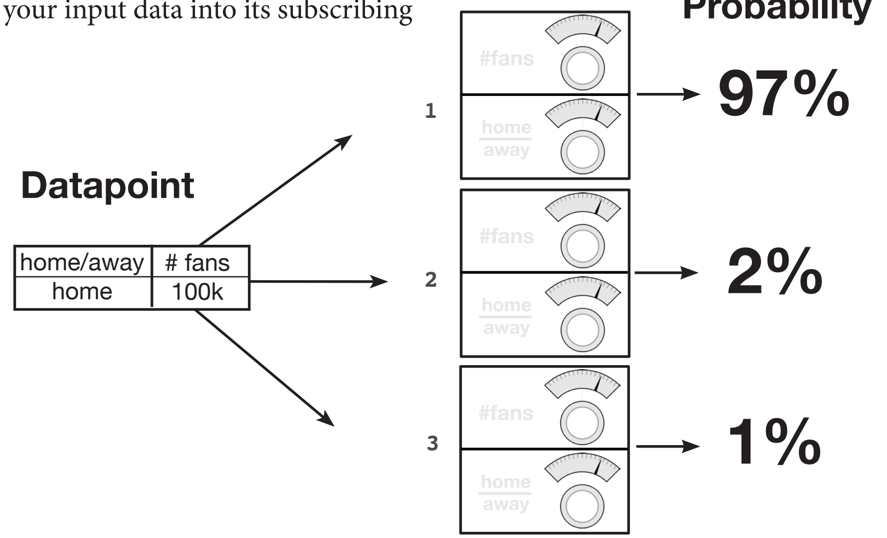
# Unsupervised Parametric Learning

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

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

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

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



# Conclusion

**Deep Learning is Parametric**

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

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

# Introduction to Neural Prediction
## Forward Propagation
# 3

## IN THIS CHAPTER


A Simple Network Making a Prediction

What is a Neural Network and what does it do?

Making a Prediction with Multiple Inputs

Making a Prediction with Multiple Outputs

Making a Prediction wtih Multiple Inputs and Outputs

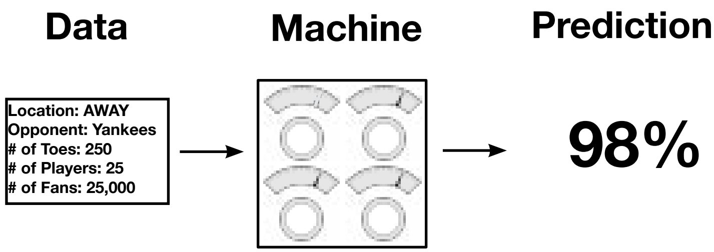
Predicting on Predictions


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

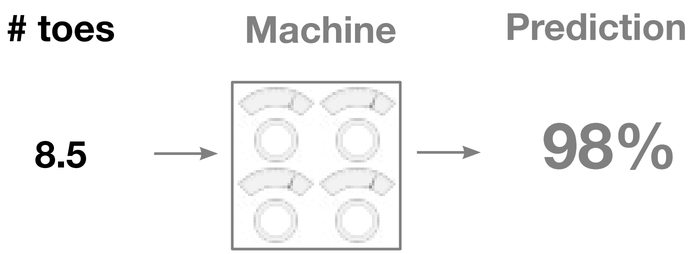
# Step 1: Predict

### 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.

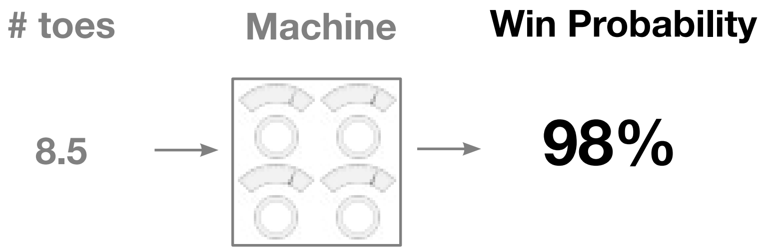



In this chapter, we're going to learn more about what these 3 different parts of a neural network prediction really look like under the hood. Let's start with the first one, the Data. In our first neural network, we're going to predict one datapoint at a time, like so.

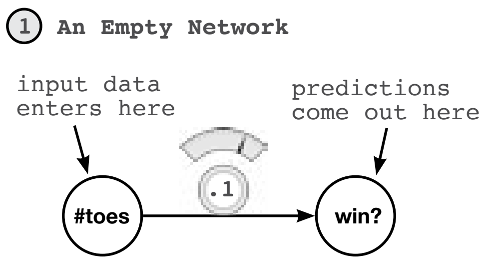



Later on, we will find that the "number of datapoints at a time" that we want to process will have a significant impact on what our network looks like. You might be wondering, "how do I choose how many datapoints to propagate at a time?" The answer to this question is based on whether or not you think 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".




Ok, so now that we know that we want to take one input datapoint and output one prediction, we can create our neural network. Now, we've had this diagram that has 4 knobs. As it turns out, the first neural network that we're going to build has one "knob" going from each simultaneous input to each simultaneous output. Since we only have one of each, we only have one "knob". These "knob"s are actually called "weight"s, and we will refer to them as such from here on out. So, without further adue, here's our first neural network.

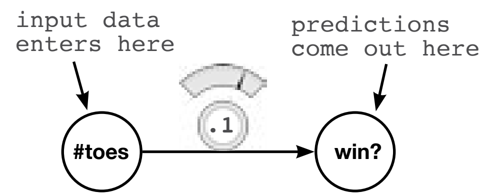



This network takes in one datapoint at a time (average number of toes on the baseball team) and outputs a single prediction (whether or not it thinks the team will win).

# A Simple Neural Network Making a Prediction

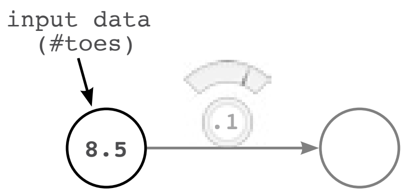
**Let's start with the simplest neural network possible.**

**1 An Empty Network**



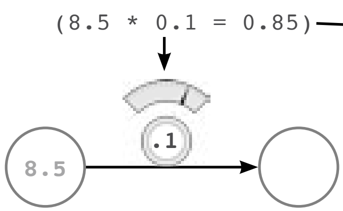

```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

**2 Inserting One Input Datapoint**



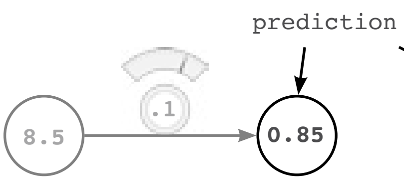

```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```

**3 Multiplying Input By Weight**




```
def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

**4 Depositing Prediction**




```
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

```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

how we use the network to predict something

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

input = number_of_toes[0]

pred = neural_network(input,weight)
```

You just made your first neural network and used it to predict! Congratulations! The last line prints out 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 vise versa.

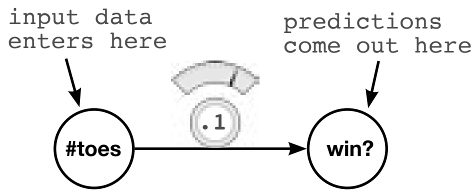
**How does the network learn?**
Trial and error! First, it tries to make a `prediction`. Then, it sees whether it was too high or too low. Finally, it changes the `weight` (up or down) to predict more accurately the next time it sees the same `input`.

# 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, 0.5) 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*.

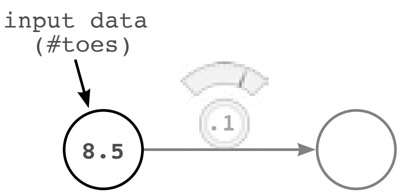
**1 An Empty Network**




```
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.

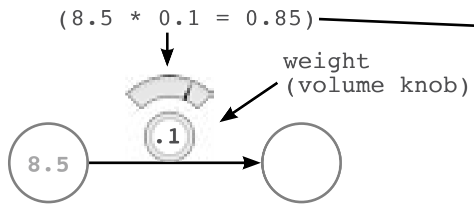
**2 Inserting One Input Datapoint**




```
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.

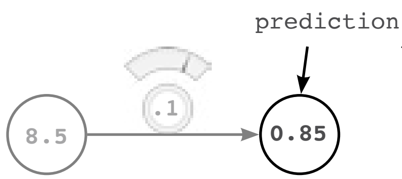
③ **Multiplying Input By Weight**




```
def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

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!

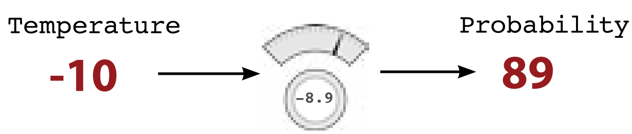
④ **Depositing Prediction**




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

input = number_of_toes[0]

pred = neural_network(input,weight)
```

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 "probably that people will wear coats today", if the temperature was -10 degrees celius, then a negative weight would predict a high probability that people would wear coats today.



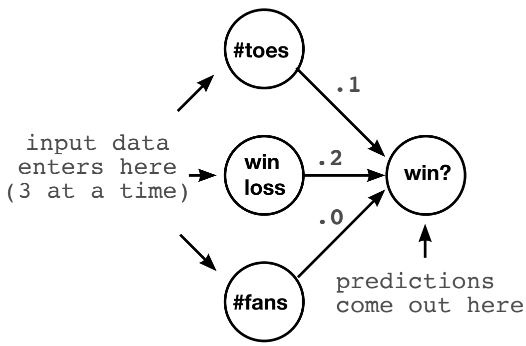
# Making a Prediction with Multiple Inputs

**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!

---

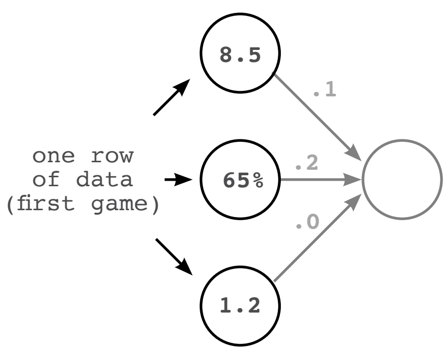
**1 An Empty Network With Multiple Inputs**




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

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

---

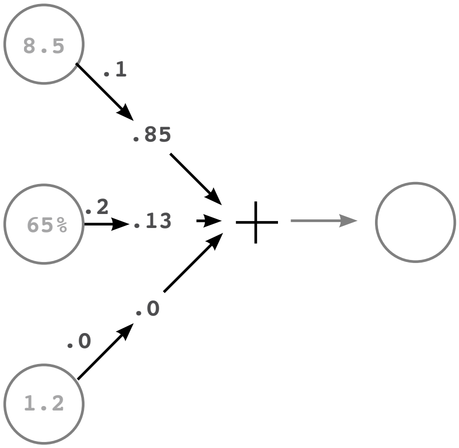
**2 Inserting One Input Datapoint**




```
/* 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)
```

---

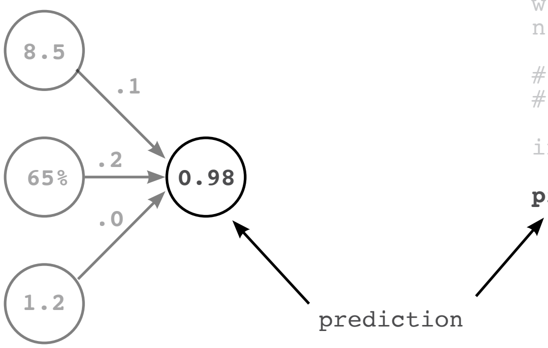
**3 Perform a Weighted Sum of Inputs**




```
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**



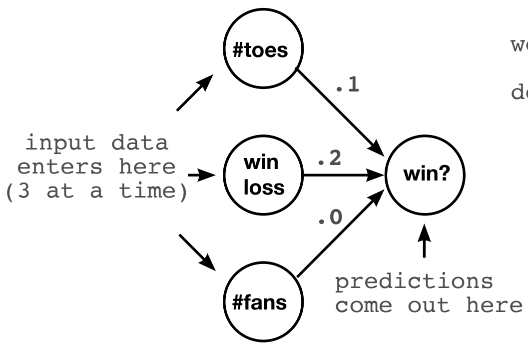

```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

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

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

pred = neural_network(input,weight)
```

# 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 this case, 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**



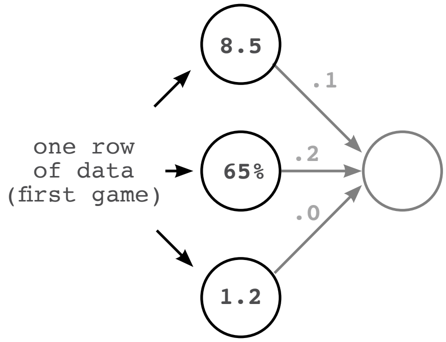

```
weights = [0.1, 0.2, 0]

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 it's 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.

**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)
```

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 it's 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.

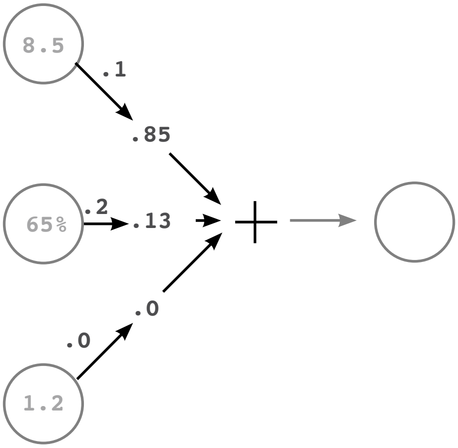
**Challenge: Vector Math**

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:

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

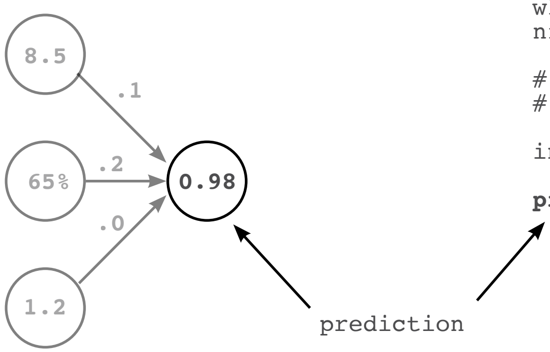
```
weights = [ 1, 0, 1] => if input[0] OR input[2]

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.**

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 still 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 adue, here's the same code in numpy.

**Previous Code**

```
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

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)
```

**Numpy Code**

```
import numpy as np

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

def neural_network(input, weights):

    pred = input.dot(weights)

    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 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.

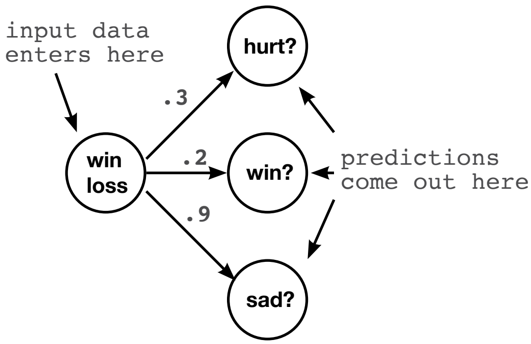
# Making a Prediction with Multiple Outputs

**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**



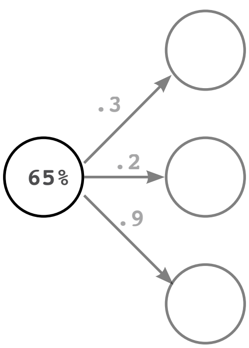

```
/* instead of predicting just
whether the team won or lost,
now we're also predicting whether
they are happy/sad AND the percentage
of the team that is hurt. We are
making this prediction using only
the current win/loss record */

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.

---

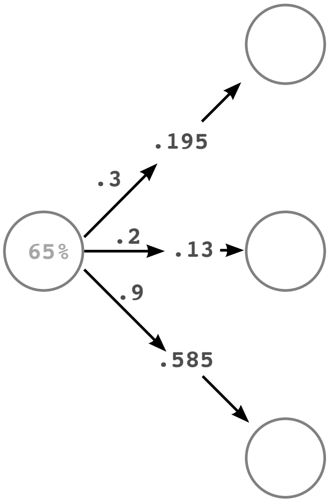
**(2) Inserting One Input Datapoint**




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

input = wlrec[0]

pred = neural_network(input,weight)
```

---

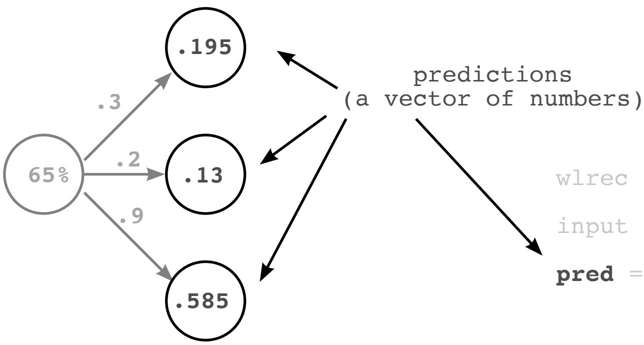
## ③ Perform an Elementwise Multiplication




```
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
( 1.20   *   0.9 )  =   0.585   = sad prediction
```

## ④ Deposit Predictions



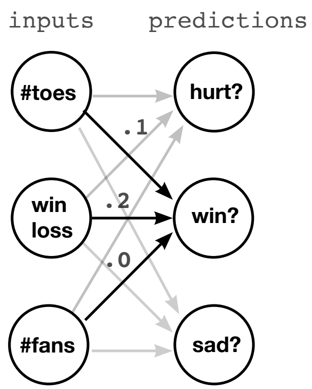

```
wlrec = [0.9, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weight)
```

# Predicting with Multiple Inputs & Outputs

**Neural networks can predict multiple outputs given multiple inputs.**

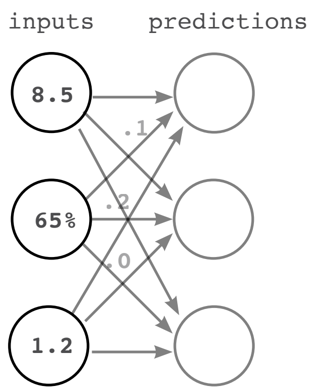
Finally, the way in which we built a network with multiple inputs or outputs can be combined together to build a network that has both multiple inputs AND multiple outputs. Just like before, we simply have a weight connecting each input node to each output node and prediction occurs in the usual way.

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



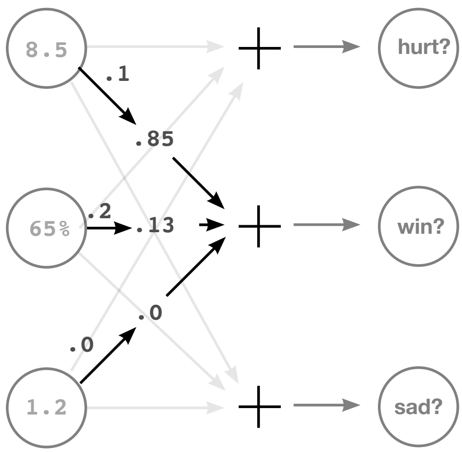

```
                 #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

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)
```

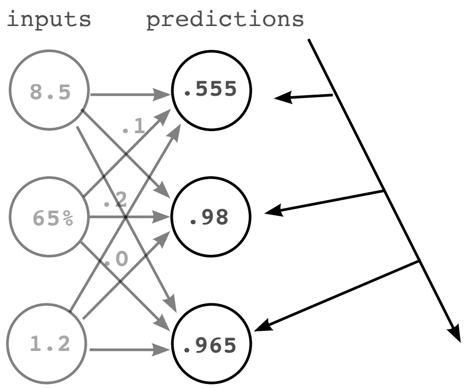
## 3 For Each Output, Perform a Weighted Sum of Inputs




```
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 = 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




```
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)
```

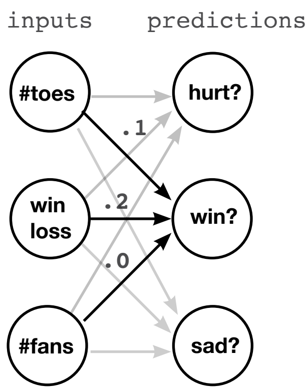
# Multiple Inputs & Outputs - How does it work?

**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 the 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 independentant weighted sums of the input. Each output node takes its own weighted sum of the input and makes a prediction.

---

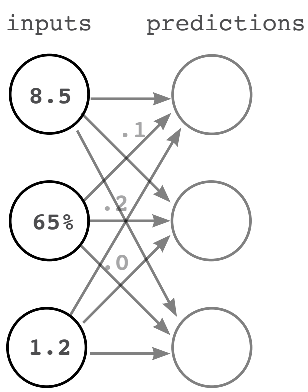
**(1) An Empty Network With Multiple Inputs & Outputs**




```
                #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
```

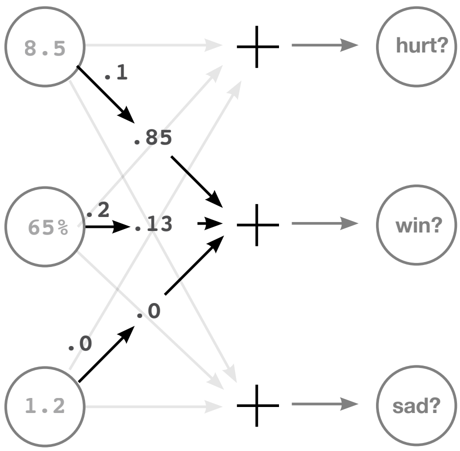
---

**(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.

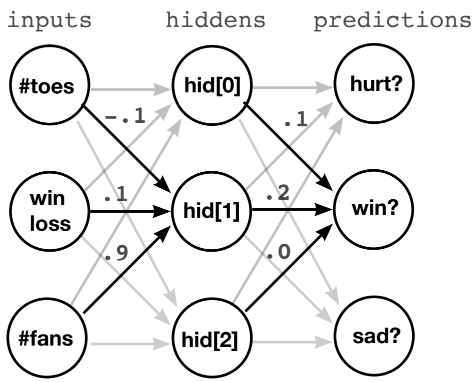
# Predicting on Predictions

## 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.

---

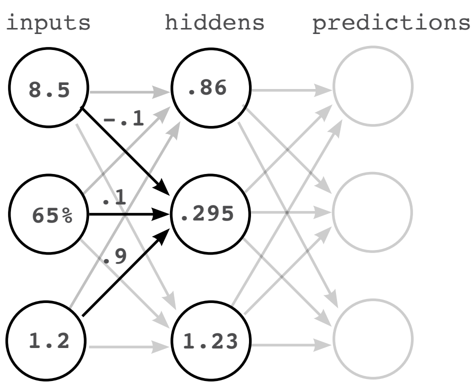
**1 An Empty Network With Multiple Inputs & Outputs**




```
           #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
```

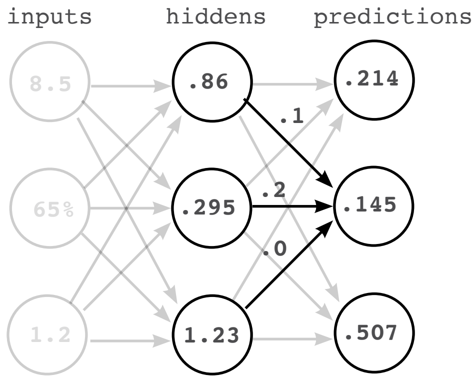
---

**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)**




```
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 to 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 he wants 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 4 columns
f = np.zeros((1,3)) # matrix with 4 rows & 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 2 rows and 4 columns
i = np.zeros((5,6)) # matrix with 4 rows & 3 columns

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


h = np.zeros((5,4)) # matrix with 2 rows and 4 columns
i = np.zeros((5,6)) # matrix with 4 rows & 3 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**

Do neural networks make accurate predictions?

Why measure error?

Hot and Cold Learning

Calculating both direction and amount from error

Gradient Descent

Learning is Just Reducing Error

Derivatives and how to use them to learn

Divergence and Alpha

> "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 to simple to be important), but 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"-esque 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.**

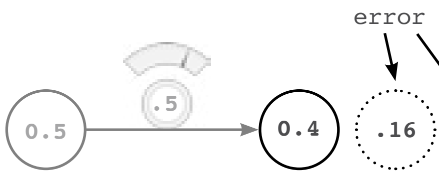
Learning is all about "error attribution", or the art of figuring out how each weight played its part in creating error. It's kindof the "blame game" of Deep Learning. In this chapter, we will spend a great number of pages learning the most popular version of Deep Learning "blame game" called Gradient 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 "0.16".




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

pred = input * knob_weight

error = (pred - goal_pred) ** 2

print(error)
```

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

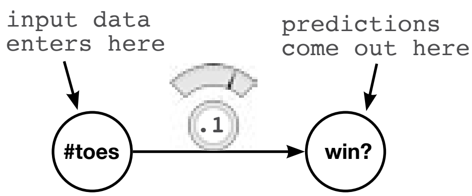
At the end of the day, learning is really about one thing, adjusting our **`knob_weight`** either up or down so that our **`error`** reduces. If we keep doing this and our **`error`** goes to 0, we are done learning! So, how do we know whether to turn the knob up or down? Well, we try *both up and down* 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.

---

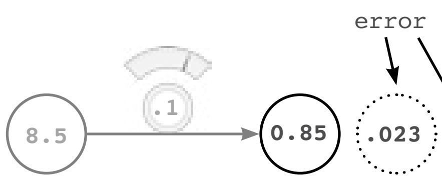
**① An Empty Network**




```
weight = 0.1

lr = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

---

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



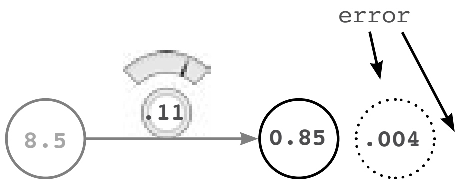

```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

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

pred = neural_network(input,weight)

error = (pred - true) ** 2
```

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.

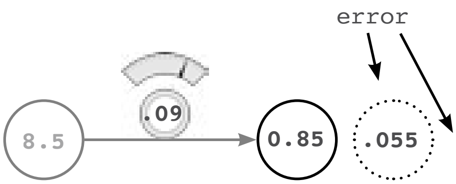
**(3) COMPARE: Making A Prediction With a Higher Weight And Evaluating Error**




```
                                               higher
lr = 0.01

p_up = neural_network(input,weight+lr)

e_up = (p_up - true) ** 2
```

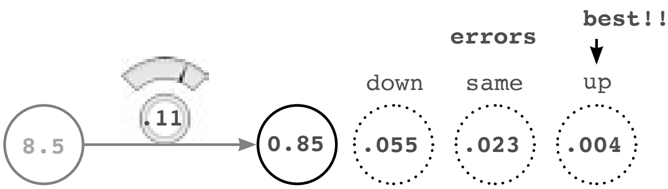
We want to move the weight so that the error goes downward, so we're going to try moving the weight up and down to see which one has the lowest error. First, we're trying moving the weight up (weight+lr).

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




```
                                              lower
lr = 0.01

p_dn = neural_network(input,weight-lr)

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

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




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

  if(e_dn < e_up):
      weight -= lr

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

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>

---

how much to move our weights each iteration

```
weight = 0.5
input = 0.5
goal_prediction = 0.8

step_amount = 0.001
```

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

```
for iteration in range(1101):

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

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

TRY UP!

```
    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 is better, go down!

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

If up is better, go up!

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

---

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

---

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

Our last step correctly predicts 0.8!

---

# Characteristics of Hot and Cold Learning

**It's simple**

Hot and Cold learning is simple. After making our prediction, we predict two more times, once with a slightly higher weight and again with a slightly lower weight. We then move the `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!!!

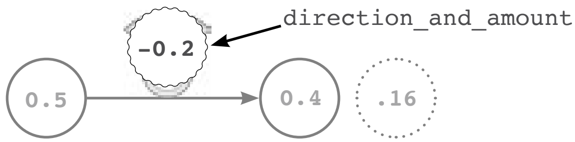
```
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
....
.... repeating infinitely...
```

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

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.




(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
    knob_weight = weight - direction_and_amount

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

What you see above is a *superior* form of learning known as **Gradient Descent**. This method allows us to (in a single line of code... seen above in **bold**) calculate both the *direction* and the *amount* that we should change our `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 vise versa. If this is a *big* number then we missed by a *big* amount, etc.

**What is "scaling, negative reversal, and stopping"?**
These three attributes have the combined affect of translating our "pure error" into "the absolute amount that we want to change our `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!**

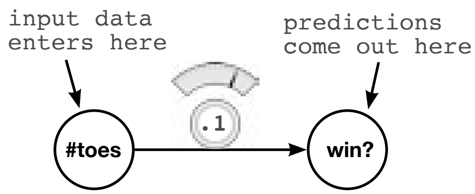
In this example, we saw Gradient Descent in action in a bit of an oversimplified environment. On the next page, we're going to see it in it's more native environment. Some terminology will be different, but we will code it in a way that makes it more obviously applicable to other kinds of networks (such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

---

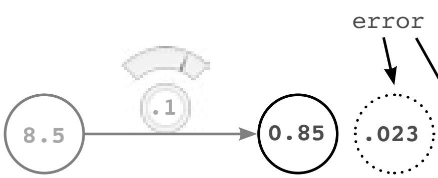
**1 An Empty Network**




```
weight = 0.1

alpha = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

---

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

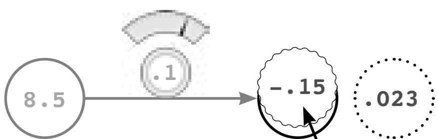
The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error"

raw error

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

---

**3 COMPARE: Calculating "Node Delta" and Putting it on the Output Node**




```
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
```

node delta

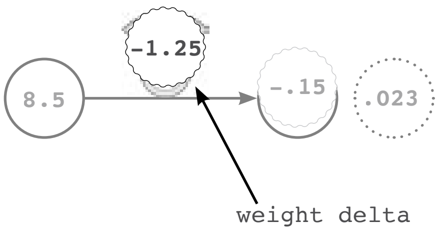
Delta is a measurement of "how much this node missed". Thus, since the true prediction was 1.0, and our network's prediction was 0.85, the network was too ***low*** by 0.15. Thus, delta is ***negative*** 0.15.

---

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**

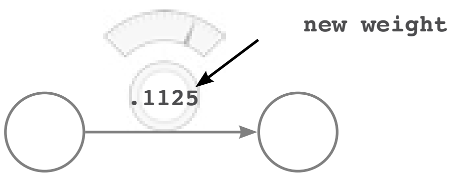



```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

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 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**




```
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:

**these lines have a *secret***

```
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 = delta * input
    weight = weight - weight_delta

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

**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

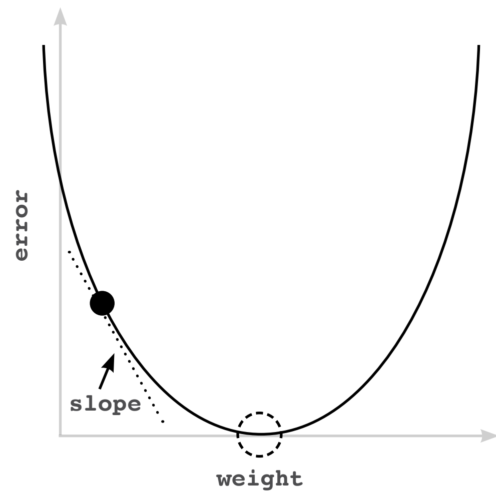
```
error = ((0.5 * weight) - 0.8) ** 2
```

**The Secret**

For any `input` and `goal_pred`, there is an *exact relationship* defined between our error and knob_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?



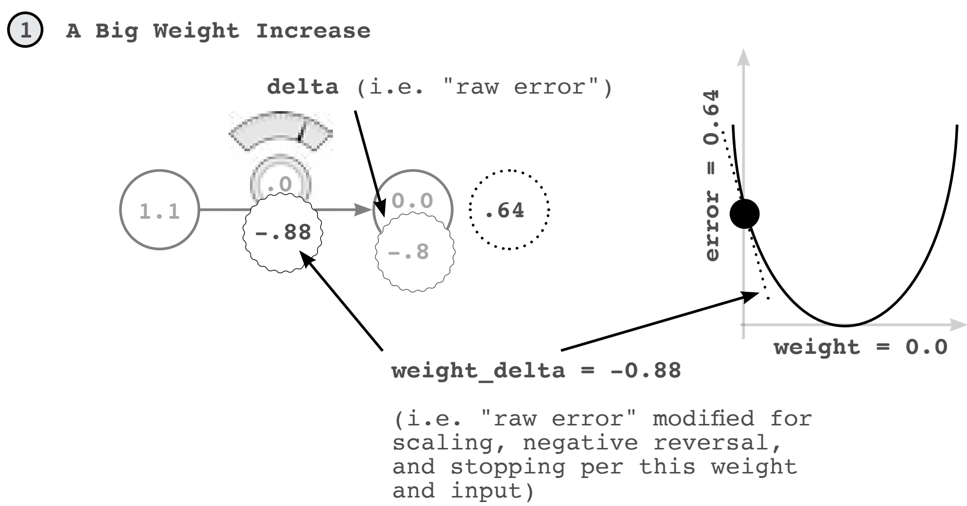

This graph represents *every value of error* for *every weight* according to the relationship in the formula above. Notice it makes a nice *bowl shape*. The black "dot" is at the point of BOTH our current `weight` and `error`. The dotted "circle" is where we want to be (`error` == 0).

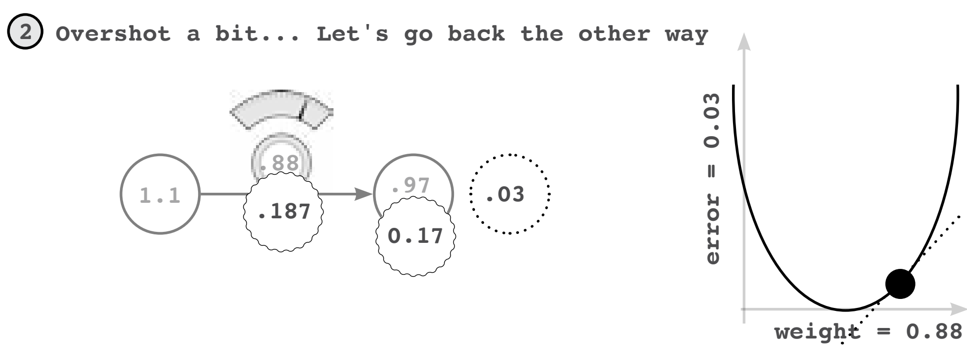
**Key Takeaway:** The *slope* points to the <u>bottom</u> of the bowl (lowest `error`) *no matter where you are in the bowl*. We can use this *slope* to help our neural network *reduce the error*.

# Let's Watch Several Steps of Learning

### Will we eventually find the bottom of the bowl?

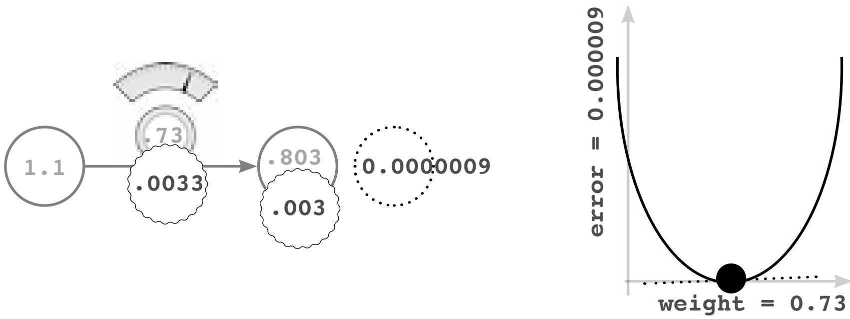
```
weight = 0.0
goal_pred = 0.8
input = 1.1

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

    print "Error:" + str(error) + " Predictio
```



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


④ Ok, we're pretty much there...



# 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.

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(prediction)
```

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`** 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 a simple if statement.

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

Now, let's go back to the formula from the previous pages, where we combined our pred and error logic. As mentioned, hey 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 a two little holes. The blue rod is sticking out of the box by 2 inches, and the red rod is sticking out of the box by 4 inches. Imagine that I told you that these rods were connected in some way, but I wouldn't tell you in what way. You had to experiment to figure it out.

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

```
red_length = blue_length * 2
```

As it turns out, there's a formal definition for "when I tug on this part, how much does this other part move". It's called a **derivative** and all it really means is "how much does rod X move when I tug on rod Y."

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

**derivative**

```
red_length = blue_length * 2
```

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

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

# Derivatives... take Two

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

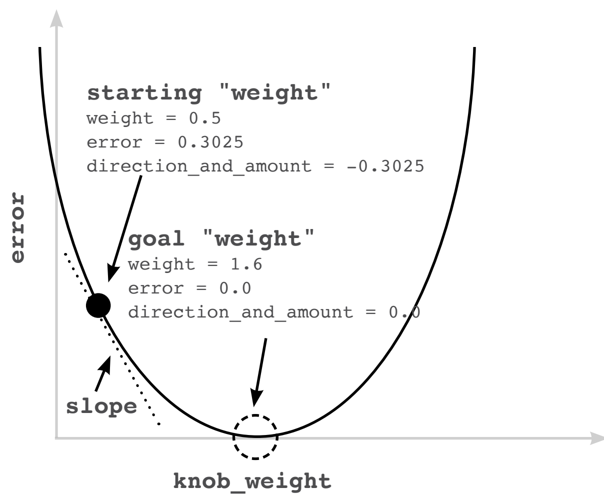
There are two ways I've heard people explain derivatives. One way is all about understanding "how one variable in a function changes when you move another variable". The other way of explaining it is "a derivative is the slope at a point on a line or curve". As it turns out, if you take a function and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

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

Now remember... our `goal_prediction` 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`.




Even now, when I look at that curve, it's easy for me to lose track of what it represents. It's actually kindof like our "hot and cold" method for learning. If we just tried *every possible value* for `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.

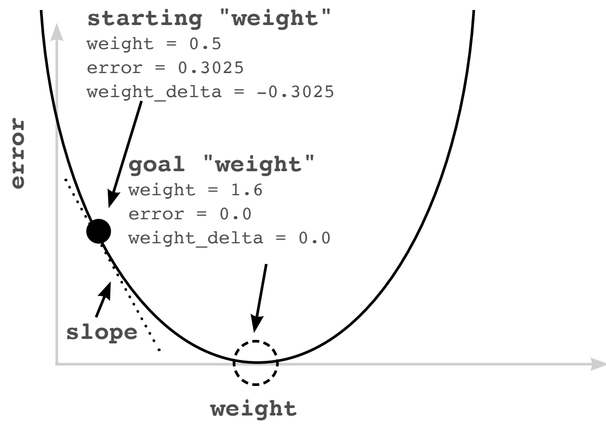
So, in this book I'm going to do what I typically do in real life (cuz i'm lazy?... i mean... efficient?) ... just look up the derivative in a reference table. All you really *need to know* is what the derivative *represents*. It's the relationship between two variables in a function so that you can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

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 vise versa. It's like gravity!

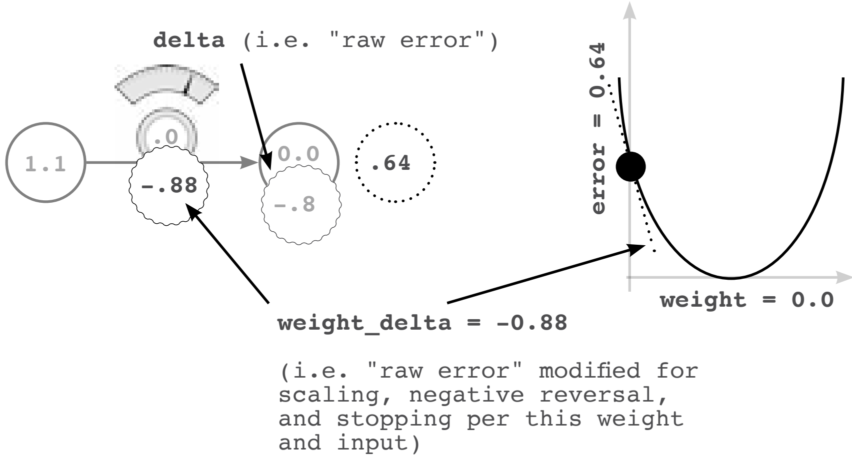
# Look Familiar?

```
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(prediction)
```

**derivative**
(i.e., how fast the
error changes given
changes in the weight)

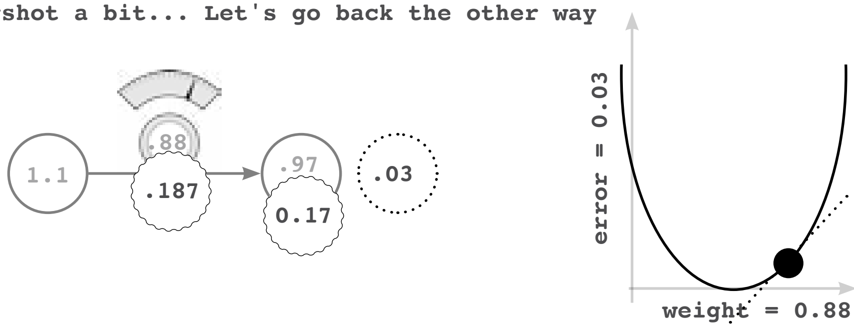
---

**1 A Big Weight Increase**




---

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




---

# Breaking Gradient Descent

### Just Give Me The Code

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

for iteration in range(20):
    prediction = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_prd
    weight_delta = input * deriv
    weight = weight - delta
```

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.

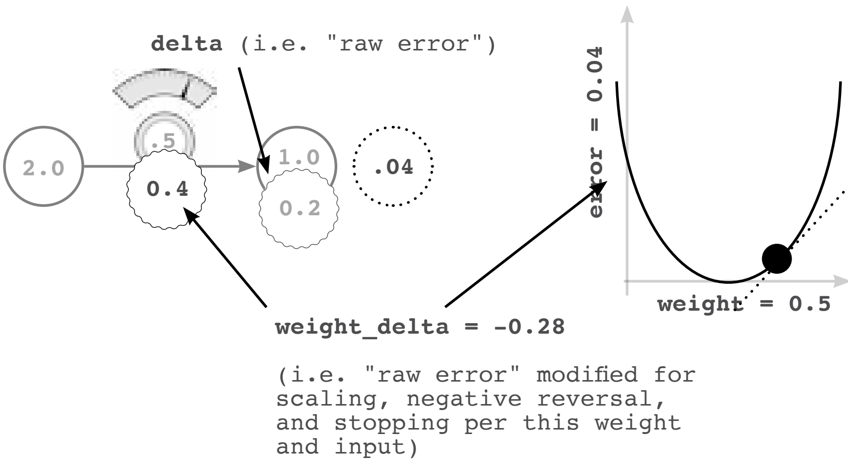
Let's try setting `input` to be equal to 2, but still try to get the algorithm to predict 0.8. What happens? Well, take a look at the output.

```
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
```

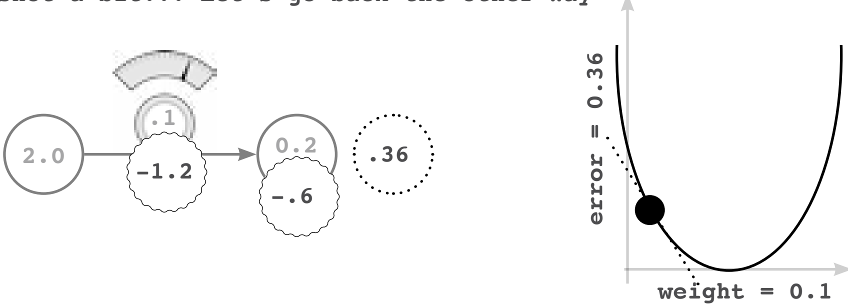
Woah! That's not what we wanted! Our predictions exploded! They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step! In other words, every update to our weight **overcorrects**! In the next section, we'll learn more about how to combat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

# Visualizing the Overcorrections

**1 A Big Weight Increase**



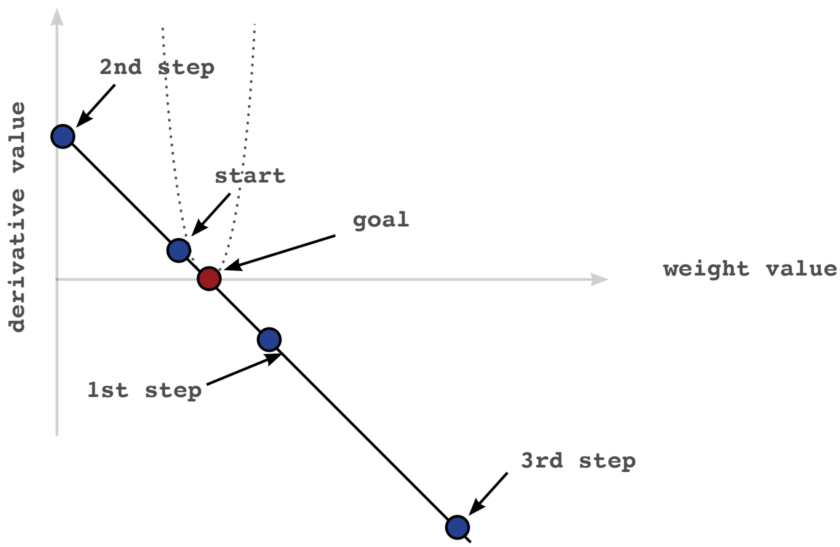

**2 Overshot a bit... Let's go back the other way**




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

## 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))
```

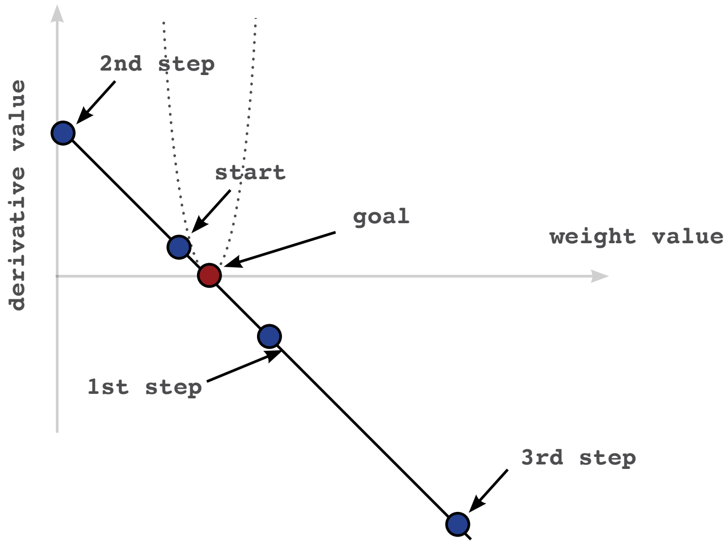
If our input is sufficiently large, this can make our weight update large even when our error is small. What happens when you have a large weight update and a small error? It overcorrects!!! If the new error is even bigger, it overcorrects even more!!! This causes the phenomenon that we saw on the previous 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 pleny of time on the newer material. It is *vitally important* that when I say something like "add your alpha parameterization to the weight update" that it is at least immediately apparent to which concepts from this chapter I'm referring.

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

# 5 Learning Multiple Weights at a Time
## Generalizing Gradient Descent

## IN THIS CHAPTER

Gradient Descent Learning with Multiple Inputs

Freezing One Weight - What does it do?

Gradient Descent Learning with Multiple Outputs

Gradient Descent Learning with Multiple Inputs and Outputs

Visualizing Weight Values

Visualizing Dot Products

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

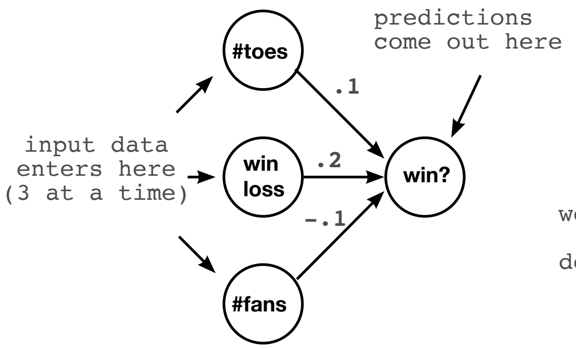
# Gradient Descent Learning with Multiple Inputs

### 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!

---

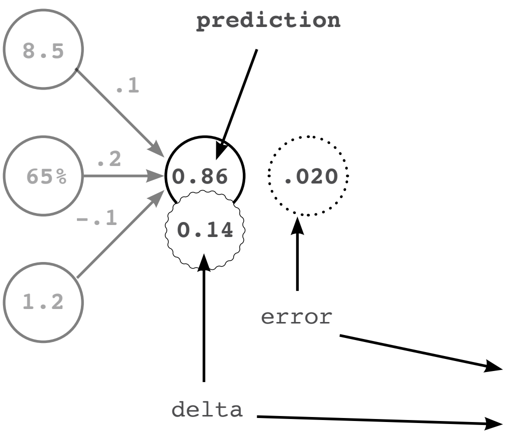
**1 An Empty Network With Multiple Inputs**




```
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
```

---

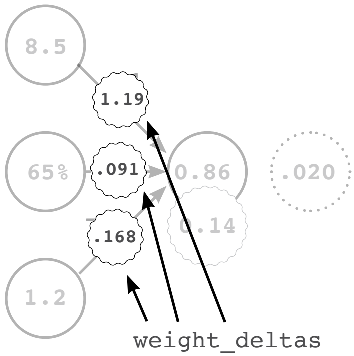
**2 PREDICT+COMPARE: Making a Prediction and Calculating Error And Delta**




```
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,weight)

error = (pred - true) ** 2

delta = pred - true
```

---

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

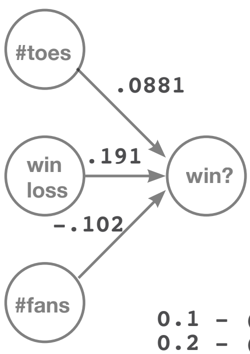



```
def ele_mul(number,vector):

    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)
```

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]
```

```
 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]
```

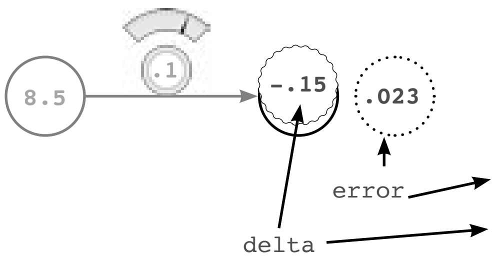
# Gradient Descent with Multiple Inputs - Explained

**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.

---

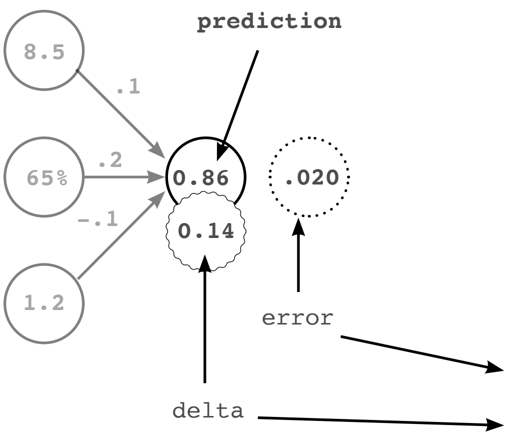
**2 Single Input: Making a Prediction and Calculating Error and Delta**




```
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?

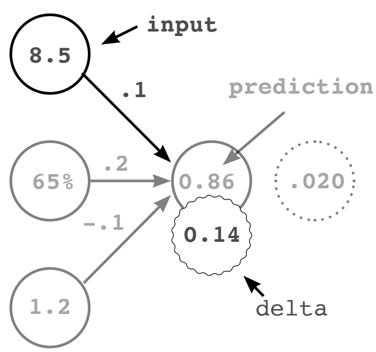
**How do we turn a single delta (on the node) into 3 weight_delta values?**

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.

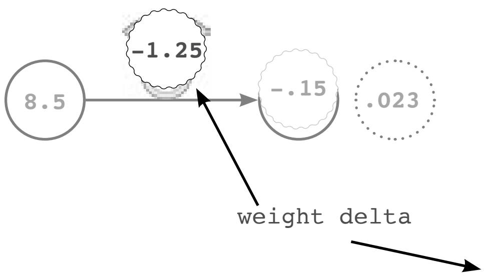
So, what did those three properties/statements really say. They all three (stopping, negative reversal, and scaling) made an observation of how the weight's role in the delta was affected by its input! Thus, each weight_delta is a sortof "input modified" version of the delta.

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**

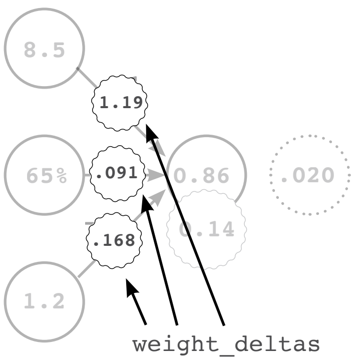



```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

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
```

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

---

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



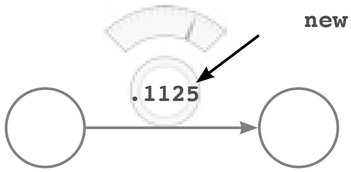

```
def ele_mul(number,vector):

    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)
```

```
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**

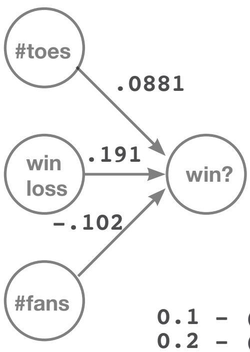



```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

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
```

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

---

**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]
```

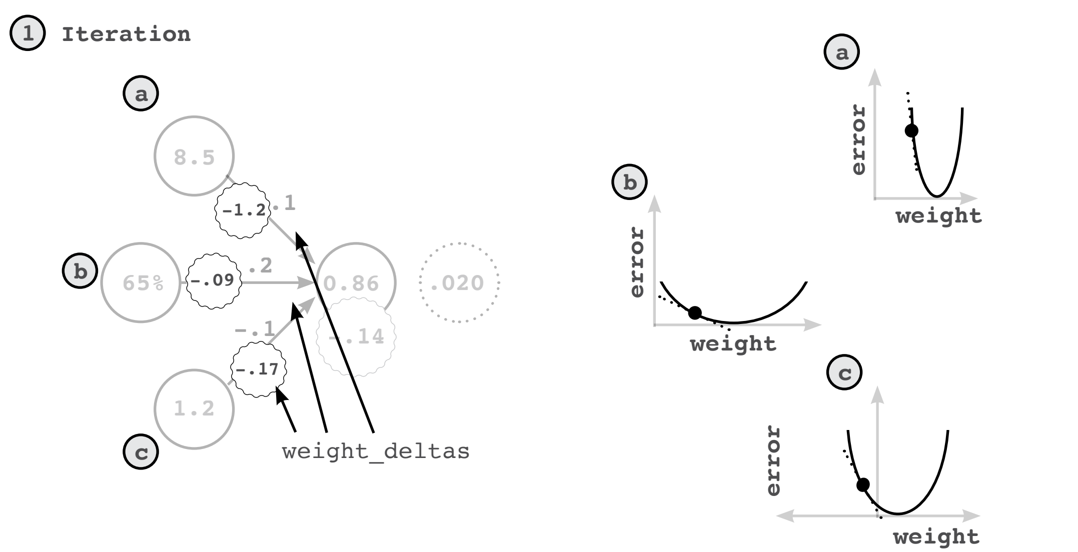
---

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 instead of just a single one.

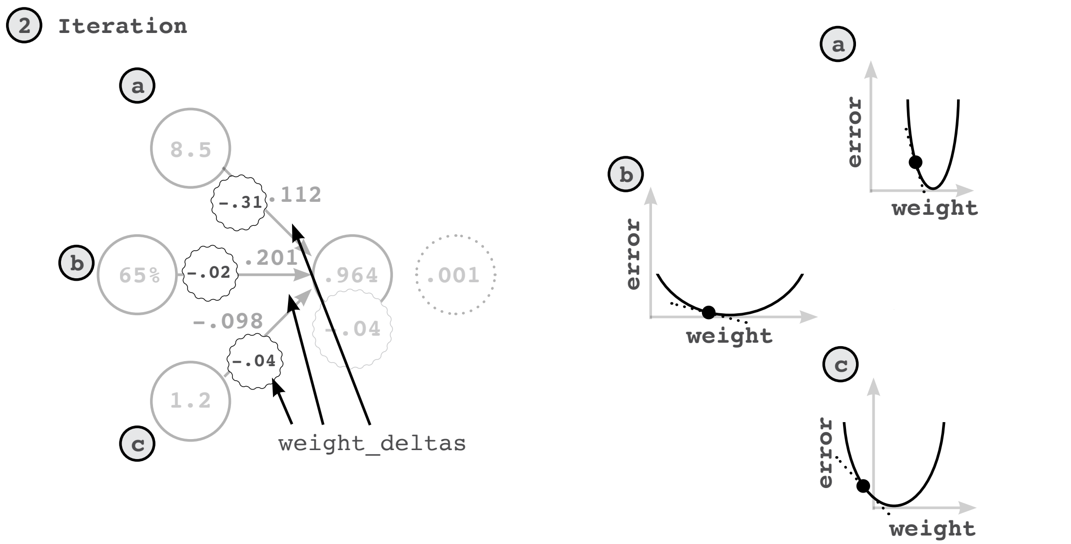
# Let's Watch Several Steps of Learning

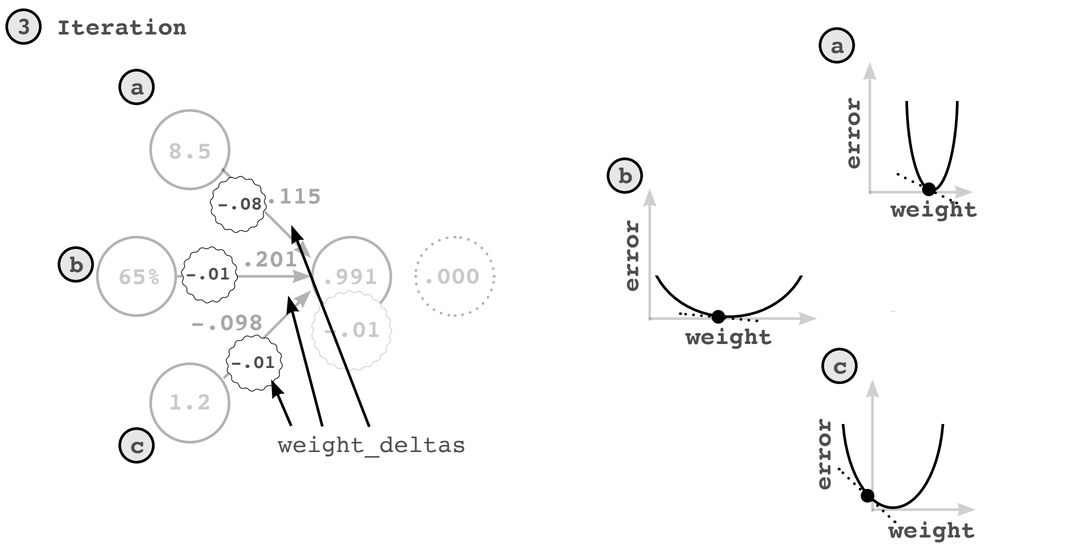
```
def neural_network(input, weights):
  out = 0
  for i in xrange(len(input)):
    out += (input[i] * weights[i])
  return out

def ele_mul(scalar, vector):
  out = [0,0,0]
  for i in xrange(len(out)):
    out[i] = vector[i] * scalar
  return out

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]

win_or_lose_binary = [1, 1, 0, 1]
true = win_or_lose_binary[0]

alpha = 0.01
weights = [0.1, 0.2, -.1]
input = [toes[0],wlrec[0],nfans[0]]
```

```
for iter in range(3):

  pred = neural_network(input,weights)

  error = (pred - true) ** 2
  delta = pred - true

  weight_deltas=ele_mul(delta,input)

  print "Iteration:" + str(iter+1)
  print "Pred:" + str(pred)
  print "Error:" + str(error)
  print "Delta:" + str(delta)
  print "Weights:" + str(weights)
  print "Weight_Deltas:"
  print str(weight_deltas)
  print

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




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.



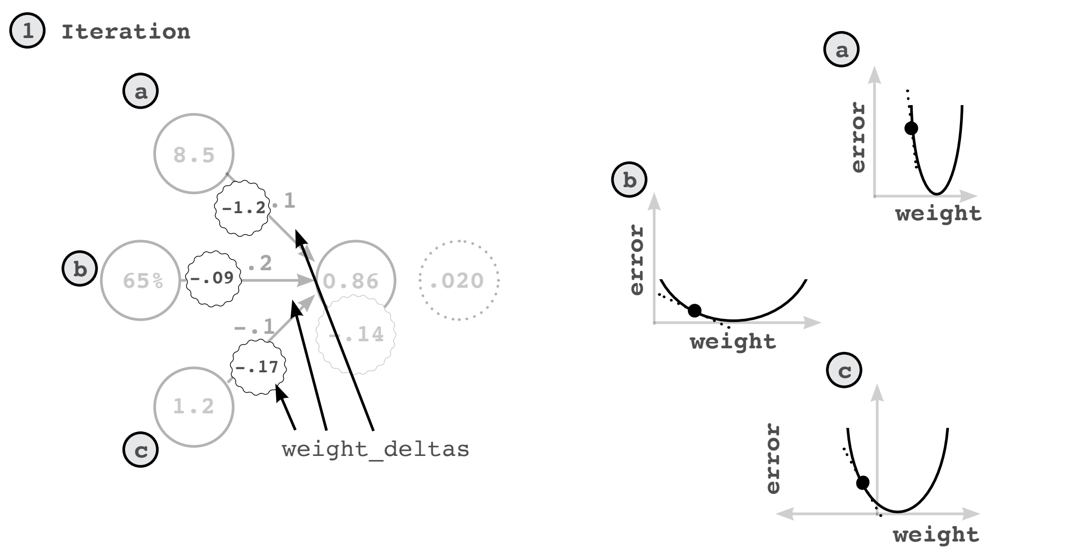

A few additional takeaways: most of the learning (weight changing) was performed on the weight with the largest input (a), because the input changes the slope significantly. This isn't necessarily advantagoues in all settings. There is a sub-field called "normalization" that helps encourage learning 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?

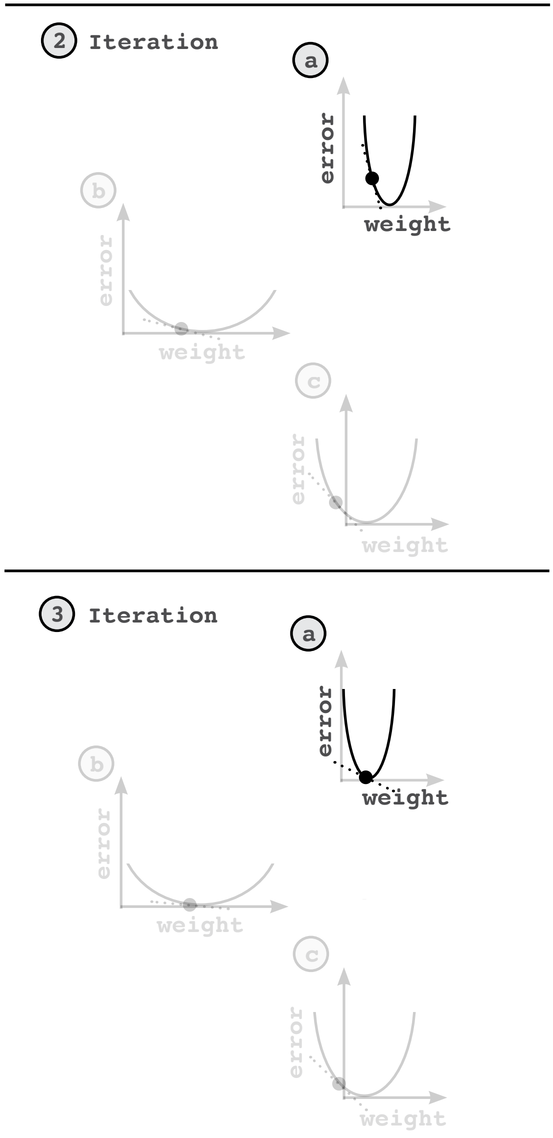
This experiment is perhaps a bit advanced in terms of theory, but I think that it's a great exercise to understand how the weights affect each other. We're going to train again, except weight a won't ever be adjusted. We'll try to learn the training example using only weights b and c (weights[1] and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

```
def neural_network(input, weights):
  out = 0
  for i in xrange(len(input)):
    out += (input[i] * weights[i])
  return out

def ele_mul(scalar, vector):
  out = [0,0,0]
  for i in xrange(len(out)):
    out[i] = vector[i] * scalar
  return out

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]

win_or_lose_binary = [1, 1, 0, 1]
true = win_or_lose_binary[0]

alpha = 0.3
weights = [0.1, 0.2, -.1]
input = [toes[0],wlrec[0],nfans[0]]
```

```
for iter in range(3):

  pred = neural_network(input,weights)

  error = (pred - true) ** 2
  delta = pred - true

  weight_deltas=ele_mul(delta,input)
  weight_deltas[0] = 0

  print "Iteration:" + str(iter+1)
  print "Pred:" + str(pred)
  print "Error:" + str(error)
  print "Delta:" + str(delta)
  print "Weights:" + str(weights)
  print "Weight_Deltas:"
  print str(weight_deltas)
  print

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



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 determiend 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.

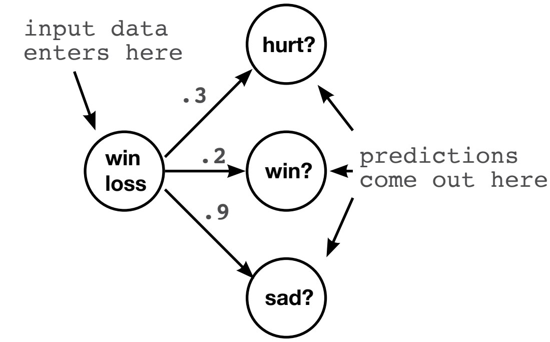
# Gradient Descent Learning with Multiple Outputs

**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.

---

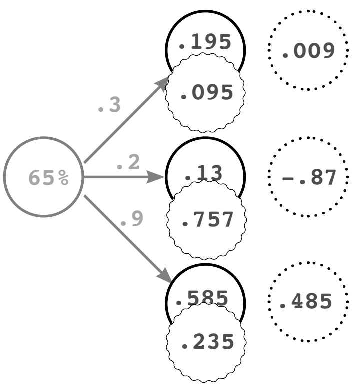
**1 An Empty Network With Multiple Outputs**




```
/* 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
```

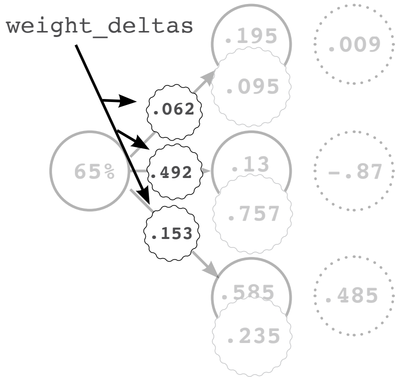
---

**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]
```

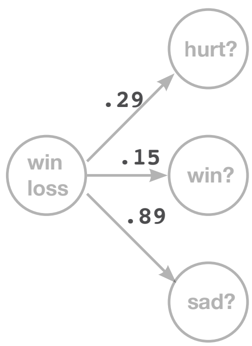
### ③ COMPARE: Calculating Each "Weight Delta" and Putting It on Each Weight




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

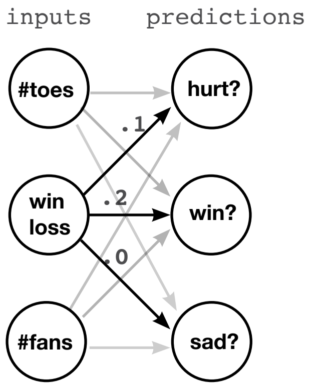



```
input = wlrec[0]
true = [hurt[0], win[0], sad[0]]
pred = neural_network(input,weight)

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

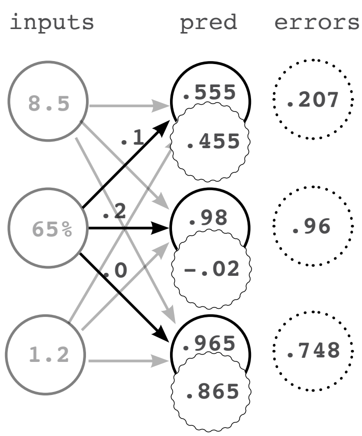
**Gradient Descent generalizes to arbitrarily large networks.**

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



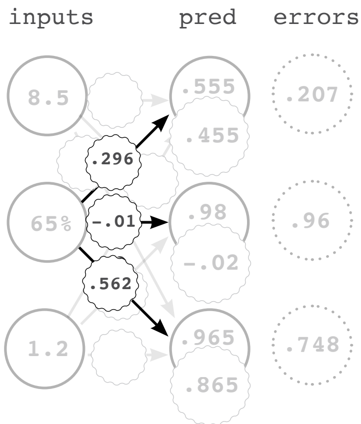

```
                 #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

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




```
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]
```

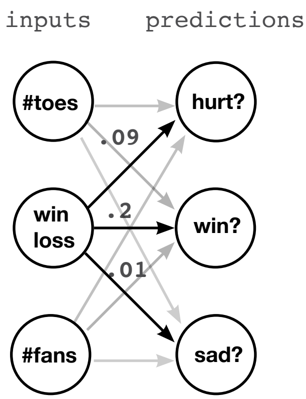
### ③ COMPARE: Calculating Each "Weight Delta" and Putting It on Each Weight




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

```
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




```
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?

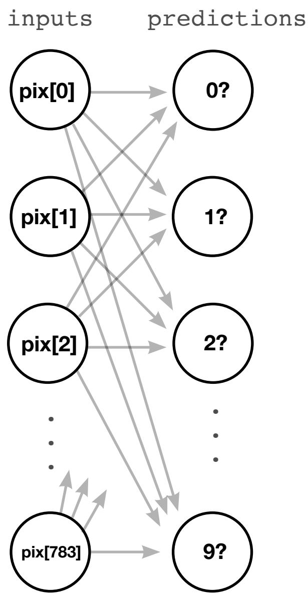
Congratulations! This is the part of the book where we move onto our first *real world dataset*. As luck would have it, it's one with historical significance!

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.

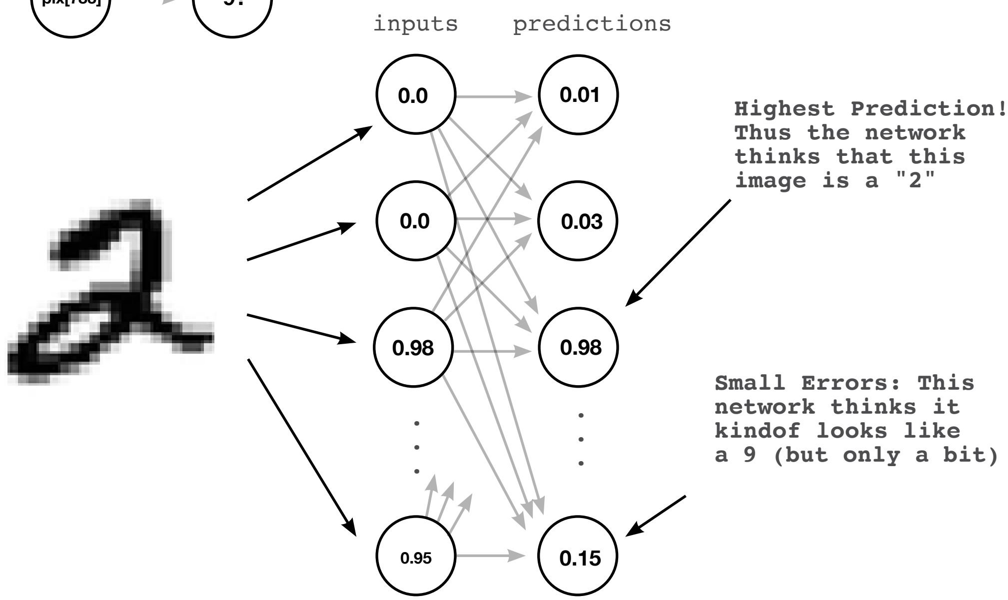
So, how do we configure our neural network to produce ten probabilties? Well, on the last page, we saw a diagram for a neural network that could take multiple inputs at a time and make multiple predictions based on that input. Thus, we should be able to simply modify this network to have the correct 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).




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).

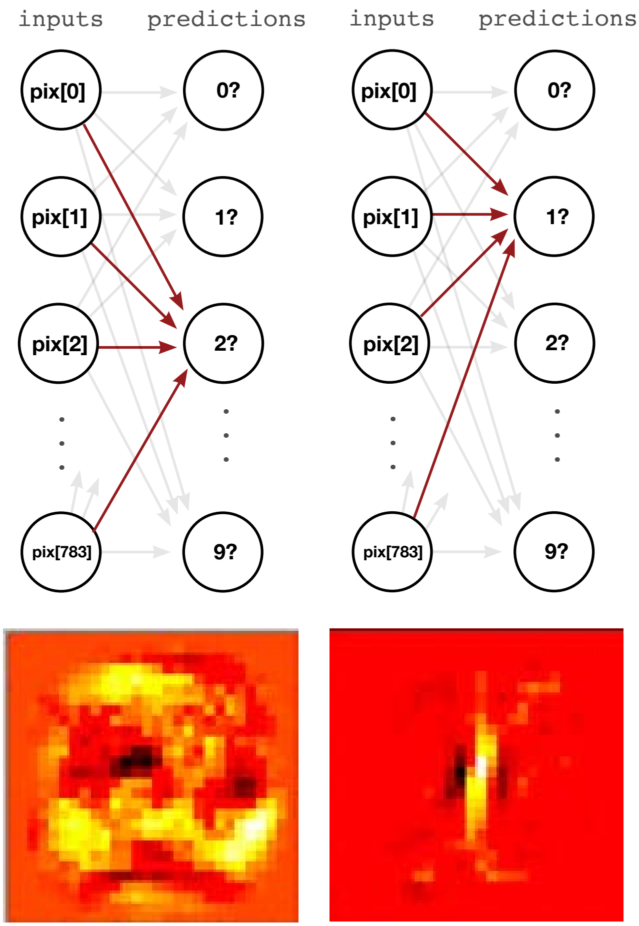
If this network was able to predict perfectly, it would take in an image's pixels (say a 2 like the one on the previous page), and predict a 1.0 in the correct output position (the third one) and a 0 everywhere else). If it was able to do this correctly for all of the images in our dataset, it would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




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:

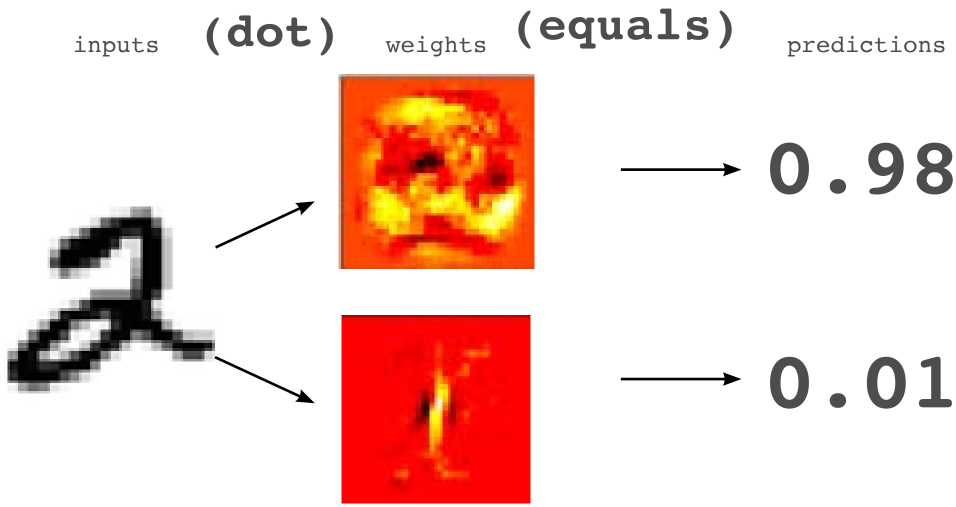
```
a = [ 0, 1, 0, 1]        score
b = [ 1, 0, 1, 0]

    [ 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)?



# 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

The Streetlight Problem

Matrices and the Matrix Relationship

Full / Batch / Stochastic Gradient Descent

Neural Networks Learn Correlation

Overfitting

Creating our Own Correlation

Backpropagation: Long Distance Error Attribution

Linear vs Non-Linear

The Secret to Sometimes Correlation

Our First "Deep" Network

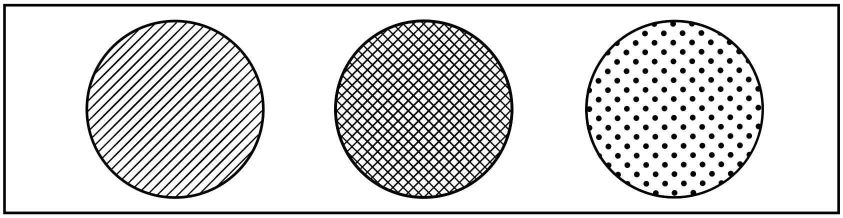
Backpropagation in Code / Bringing it all Together

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

# The Street Light Problem

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

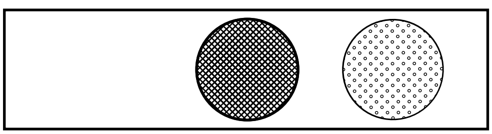
Consider yourself approaching a street corner in a foreign country. As you approach, you look up and realize that the street light is quite unfamiliar. How can you know when it is safe to cross the street?



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.

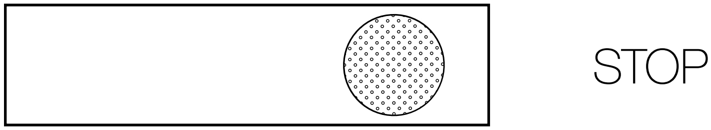
STOP

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.

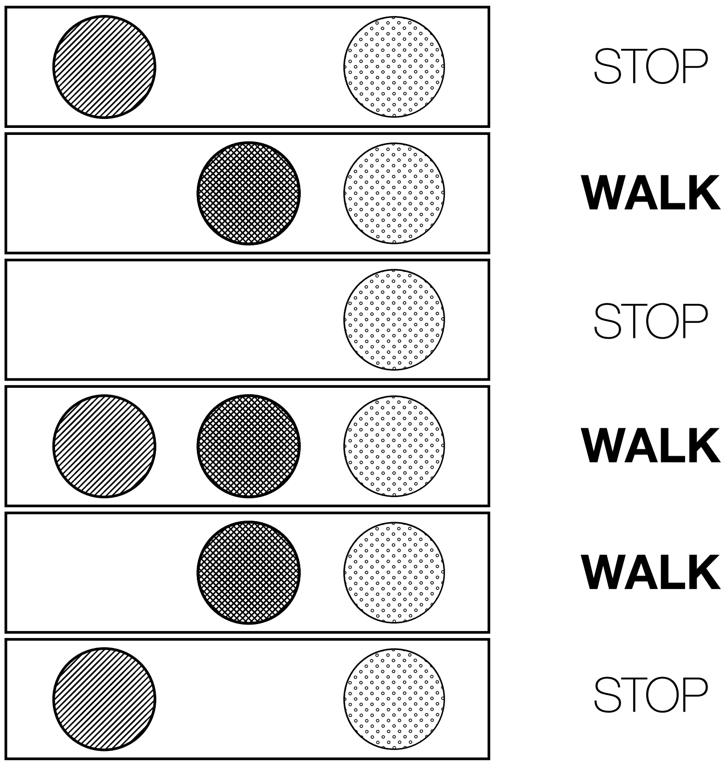


**WALK**

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.




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




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

# Preparing our Data

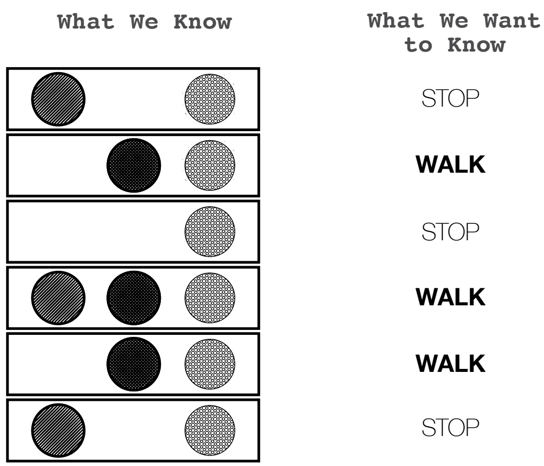
### Neural Networks Don't Read Streetlight

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

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

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

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



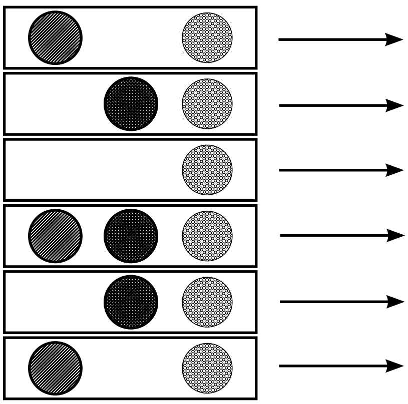

So, in order to prepare this data for our neural network, we need to first split it into these two groups (what we know and what we want to know). Note that we could attempt to go backwards if we swapped which dataset was in which group. For some problems, this works.

# Matrices and the Matrix Relationship

**Translating your streetlight into math.**

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

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).

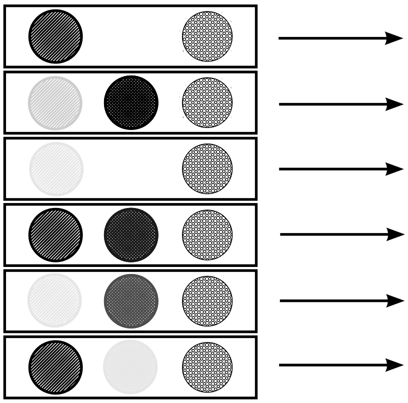
In data matrices, it is convention to give each *recorded example* a single **row**. It is also convention to give each *thing being recorded* a single **column**. This makes it easy to read.

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

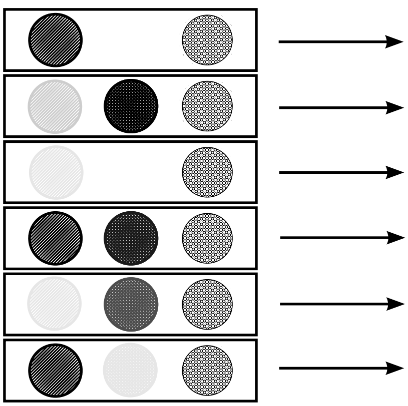


Streetlight **Matrix A**

| | | |
|---|---|---|
| .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**

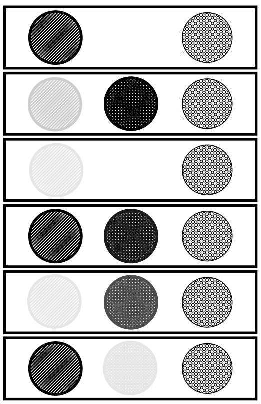
| | | |
|---|---|---|
| 9 | 0 | 10 |
| 2 | 8 | 10 |
| 1 | 0 | 10 |
| 8 | 9 | 10 |
| 1 | 7 | 10 |
| 9 | 1 | 0 |

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

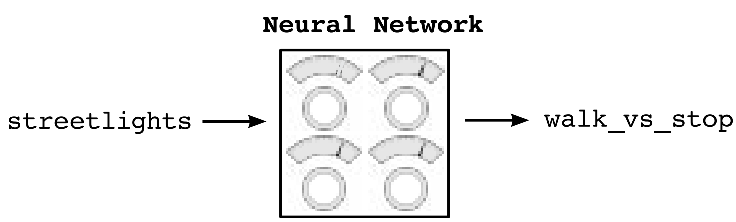
So, we've converted our streetlight pattern into a matrix (the one with just 1s and 0s). Now, we want to create that matrix (and more importantly, it's underlying pattern) in Python so that our neural network can read it. Python has a special library built just for handling matrices called **numpy**.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

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

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

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

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



# 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.

```
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.

```
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!

```
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.**

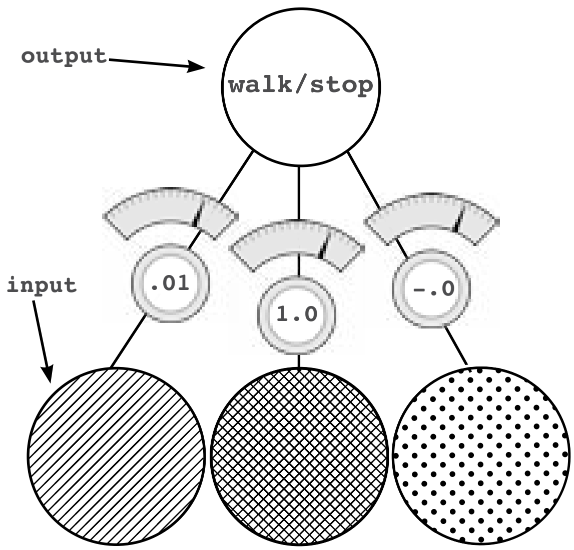
This will be covered in more detail later, but there is also a third configuration that sortof "splits the difference" between Stochastic Gradient Descent and Full Gradient Descent. Instead of updating the weights after just one or after the entire dataset of examples, you choose a "batch size" (typically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

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |

Weight Pressure

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

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

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

The "Weight Pressure" table helps describe the affect that each 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

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |

Weight Pressure

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

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

**Bottom Line**

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

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 1 | | 0 |
| 0 | 1 | 1 | → | 1 |
| 0 | 0 | 1 | | 0 |
| 1 | 1 | 1 | → | 1 |
| 0 | 1 | 1 | → | 1 |
| 1 | 0 | 1 | | 0 |

Weight Pressure

| | | | | |
|---|---|---|---|---|
| – | 0 | – | | 0 |
| 0 | + | + | → | 1 |
| 0 | 0 | – | | 0 |
| + | + | + | → | 1 |
| 0 | + | + | → | 1 |
| – | 0 | – | | 0 |

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

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

# Edge Case: Conflicting Pressure (cont.)

**It doesn't always work out like this.**

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

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 | | 1 |
| 0 | 1 | 1 | → | 1 |
| 0 | 0 | 1 | | 0 |
| 1 | 1 | 1 | → | 0 |

Weight Pressure

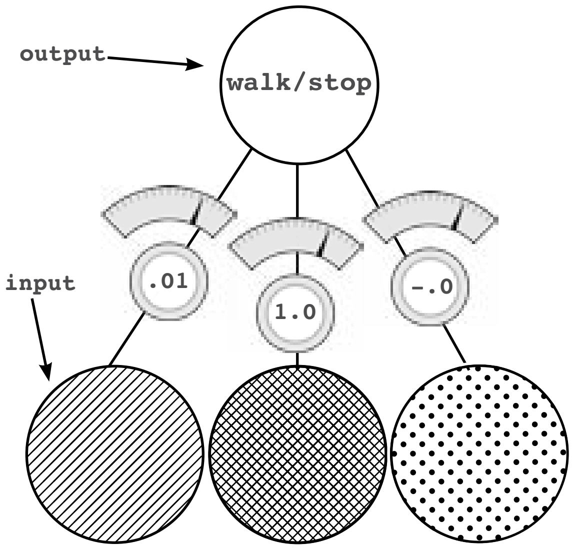
| | | | | |
|---|---|---|---|---|
| + | 0 | + | | 1 |
| 0 | + | + | → | 1 |
| 0 | 0 | – | | 0 |
| – | – | – | → | 0 |

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.



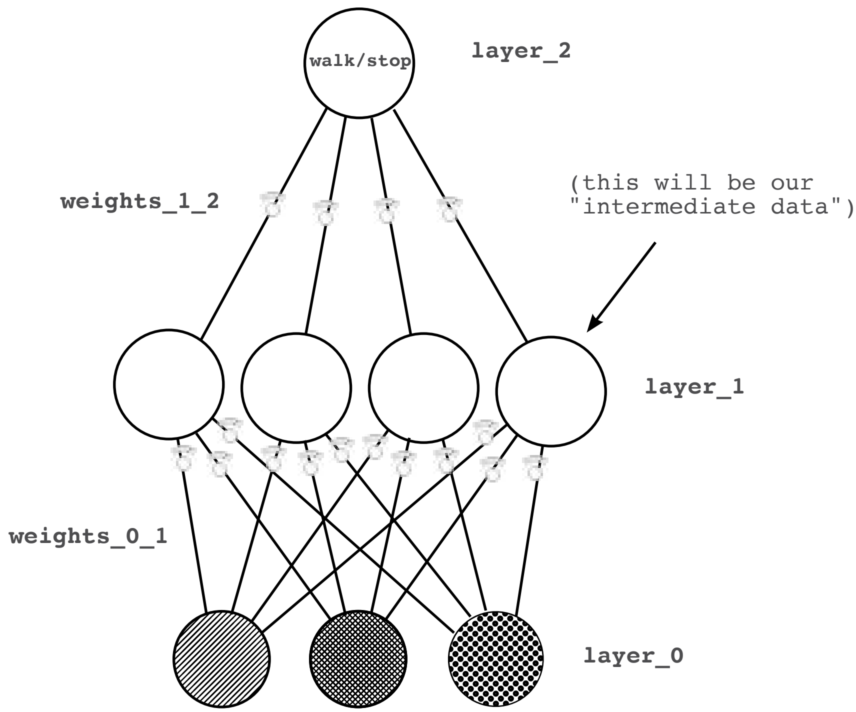

Unfortunately, we just encountered a new streetlights dataset where there isn't *any correlation* between our input and output. The solution is simple. Let's use *two* of these networks. The first one will create an *intermediate dataset* that has *limited* correlation with our output. The second will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

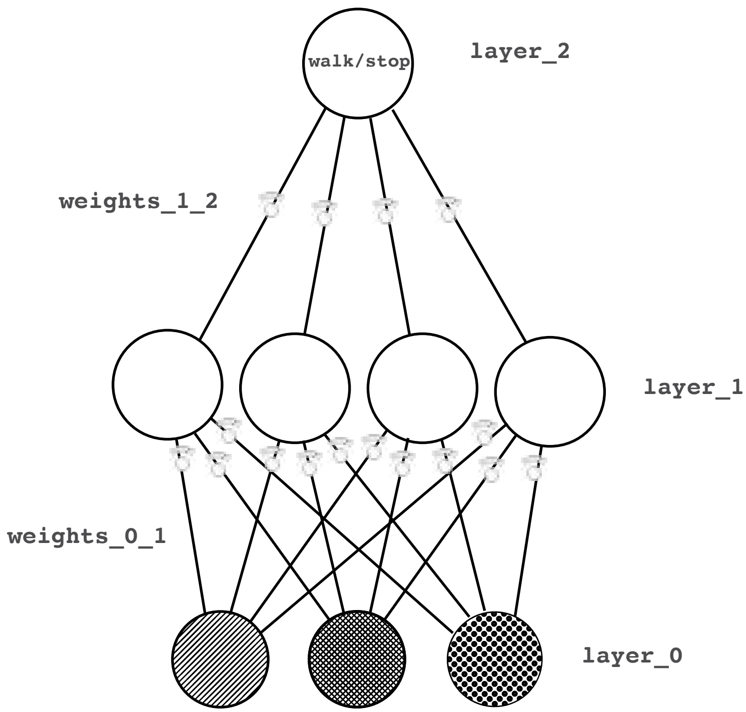



Things to notice: This network is still just a function! It still just has a bunch of weights that are collected together in a particular way. Furthermore, **Gradient Descent** still works because we can calculate how much each weight contributes to the error and adjust it to reduce the error to 0. That's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

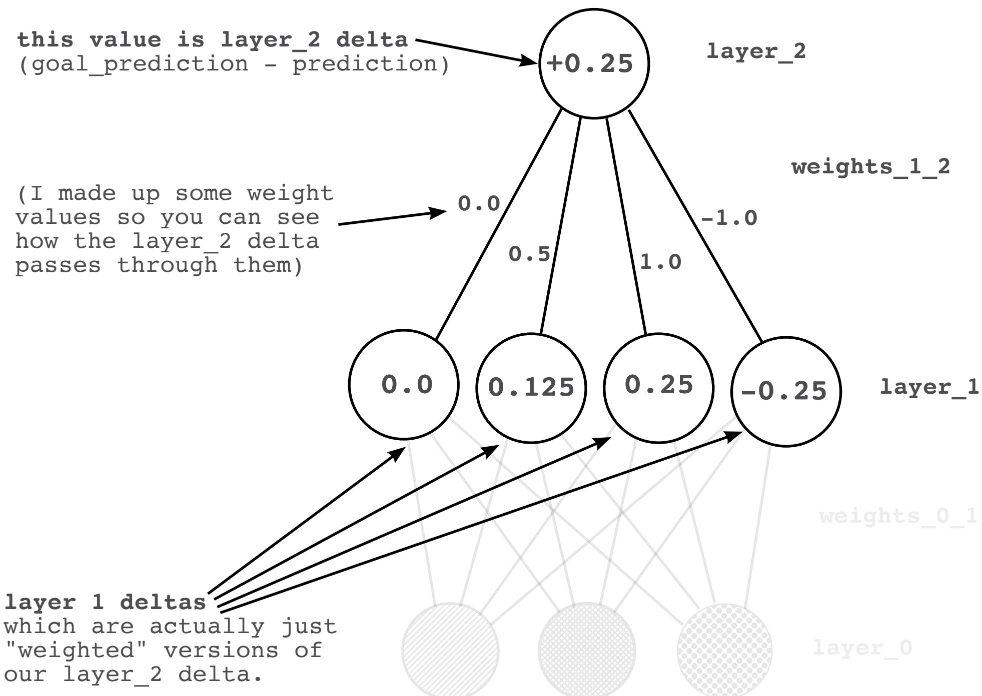
So, the part that we *don't yet understand* is how to update the weights between layer_0 and layer_1. What do they use as their error measure? If you remember from the last chapter, our *cached/normalized error measure* was called *delta*. In our case, we want to figure out how to know the *delta* values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

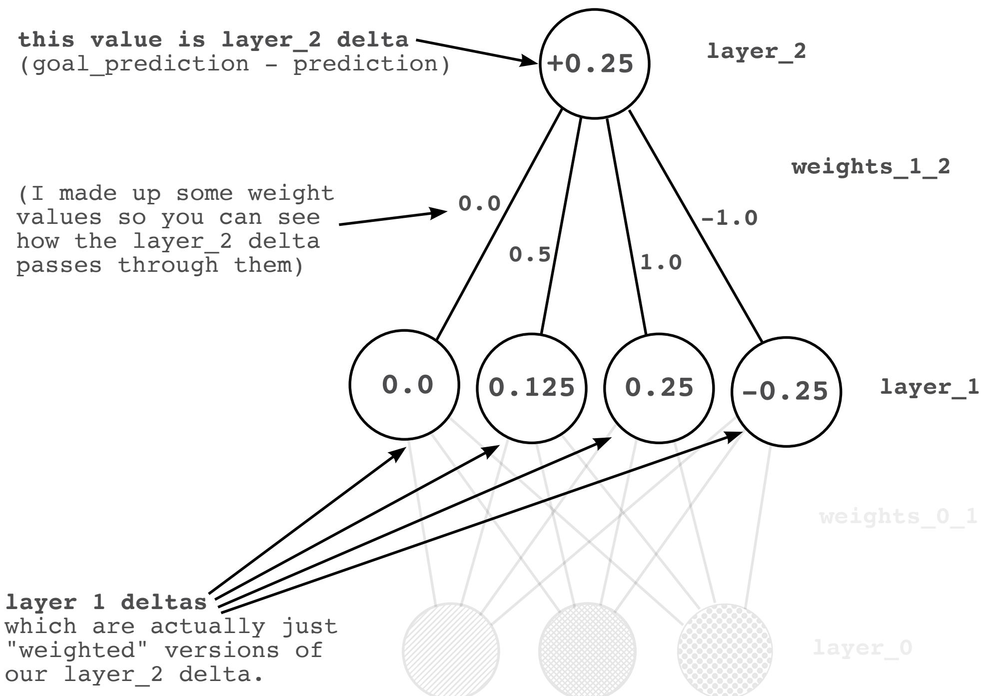
It's so simple it's almost hilarous. Our weights from layer_1 to layer_2 exactly describe how much each layer_1 neuron contributes to the layer_2 prediction. This means that those weights ALSO *exactly describe* how much each layer_1 neuron contributes to the layer_2 **error!** So, how do we use the 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 backpropagation 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

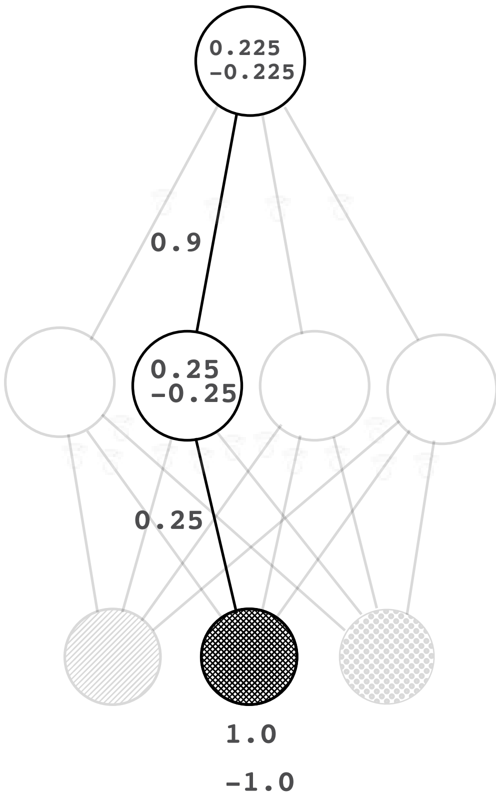
**This is probably the hardest concept in the book. Let's take it slow.**

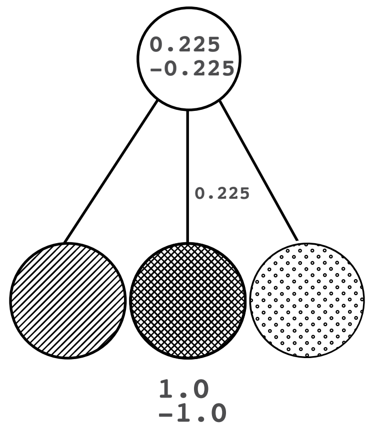
I'm going to show you a phenomenon. As it turns out, we need one more "piece" to make this neural network train. We're going to take it from two perspectives. The first is gong to show you why *the neural network can't train without it*. In other words, first I'm going to show you why our neural network 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
5 * 20 = 100
```

```
1 * 0.25 * 0.9 = 0.225
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.



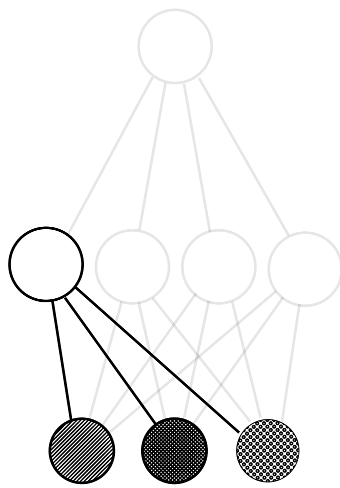

These two graphs show you *two training examples each*, one where the input is 1.0 and another where the input is -1.0. Here's the bottom line, **for any 3-layer network we create, there's a 2-layer network that has identical behavior**. It turns out that just stacking two neural nets (as we know them 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 correlation between our *input* and our *output* layers. If no correlation exists, then error will never reach 0.

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

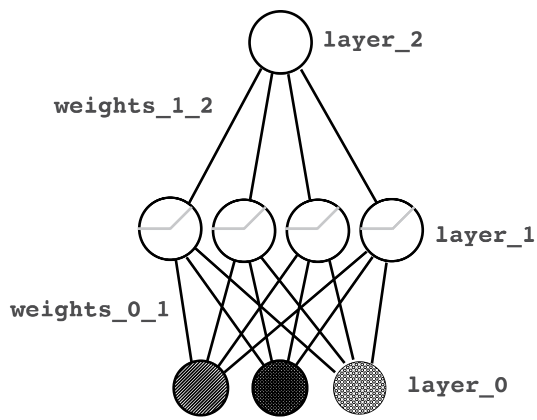
```
import numpy as np

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.



# 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.

```
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)
```

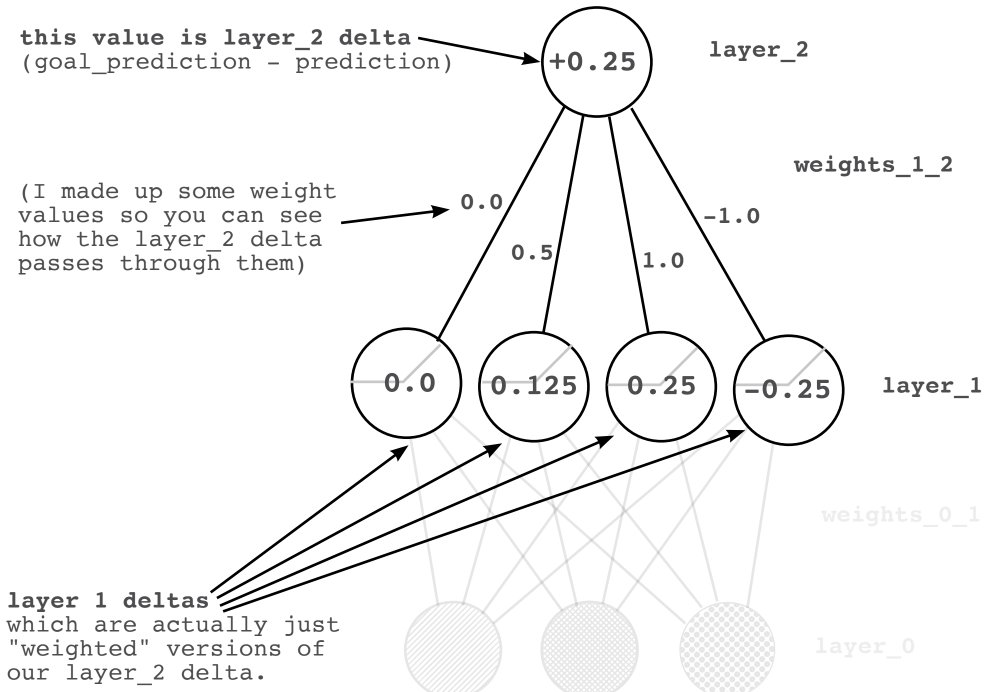
`this line computes the delta at layer_1 given the delta at layer_2 by taking the layer_2_delta and multiplying it by its connecting weights_1_2`

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)

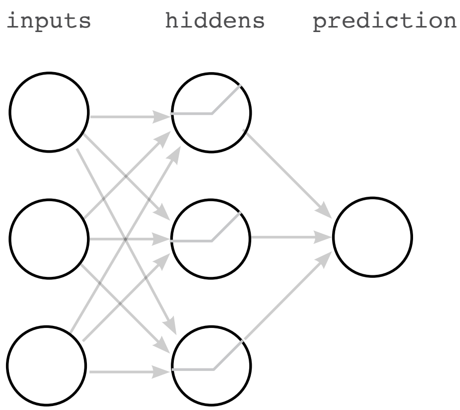
So, now that we have how much we want the final prediction to move up or down (delta), we need to figure out how much we want each middle (layer_1) node to move up or down. These are effectively *intermediate predictions*. Once we have the delta at layer_1, we can use all the same processes we used before 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.



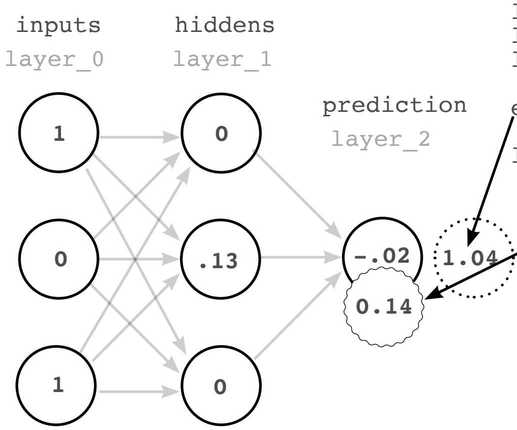
# One Iteration of Backpropagation

**① Initialize the Network's Weights and Data**



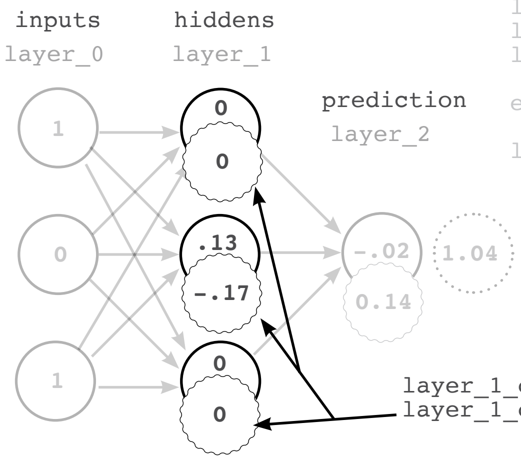

```
import numpy as np

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**



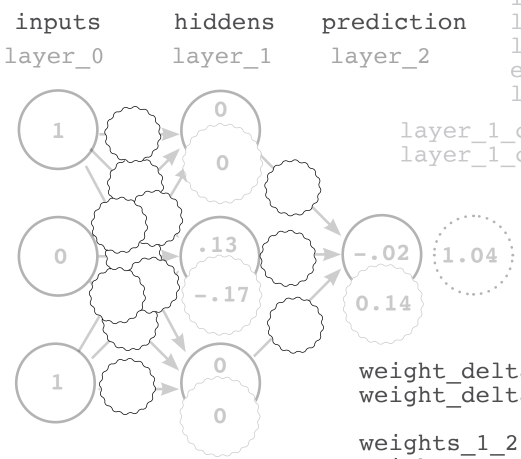

```
layer_0 = lights[0:1]
layer_1 = np.dot(layer_0,weights_0_1)
layer_1 = relu(layer_1)
layer_2 = np.dot(layer_1,weights_1_2)

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

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

**3 LEARN: Backpropagate From layer_2 to 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)

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)
```

**4 LEARN: Generate Weight Deltas and Update Weights**




```
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)**

```
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 correct 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 phenomenons 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 littel 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 layers says it should be can be determined by taking what a later layer says it shoul dbe 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.

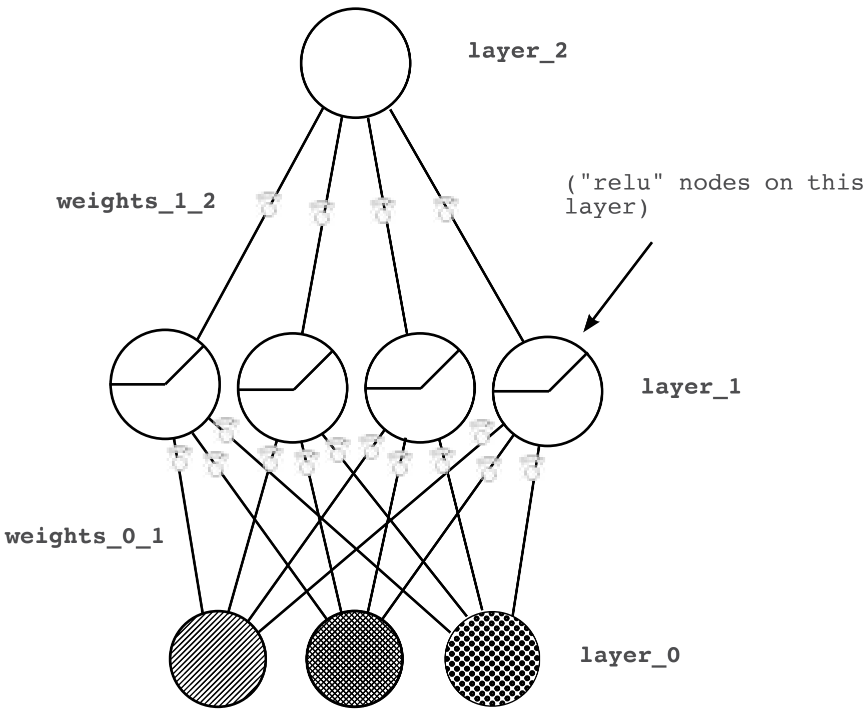
# Our Previously Overcomplicated Visualization

**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.

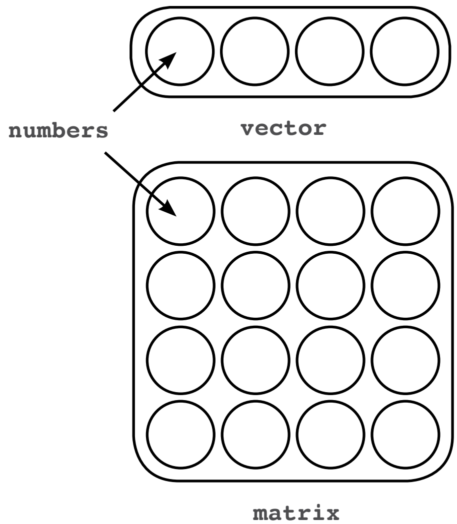
So, to refect the change in our attention, we need to update our visualization on paper. We're also going to do a few more things which will make sense later. As you know, our neural network is really just a series of weight matrices. When we're using the network, we also end up creating vectors corresponding 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.

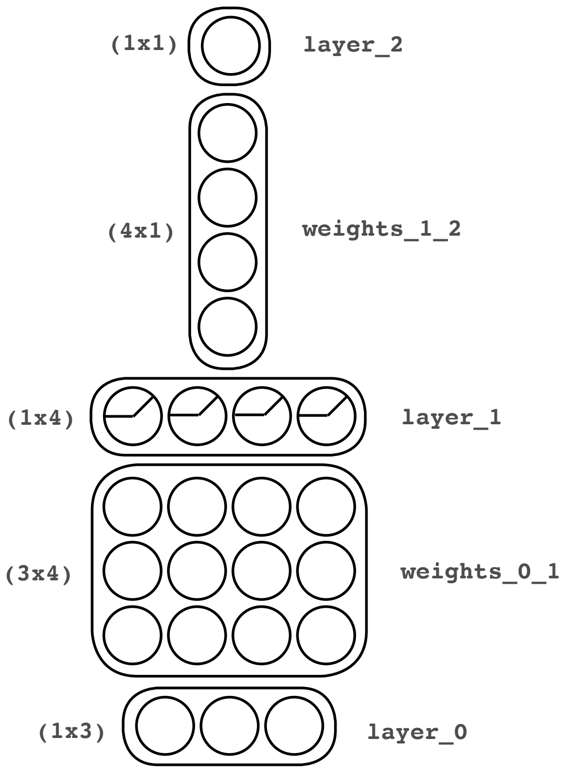


# Our Simplified Visualization

**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!



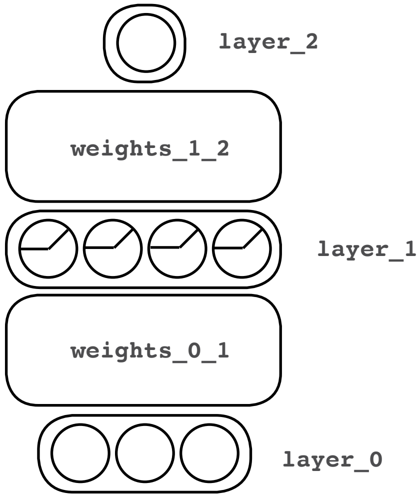




**The Big Takeaway:** This picture still gives us all the information we need to build our neural network! We know the shapes and sizes of all of our layers and matrices. All the detail from before really isn't necessary when you already know the Correlation Summarization and everything that went into 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 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 teh 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*.

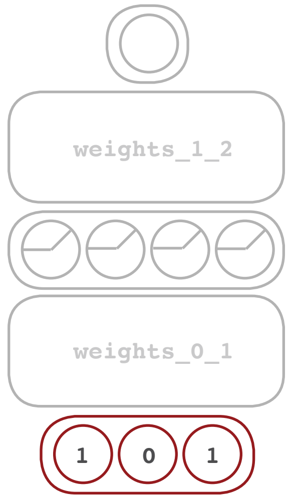
> Good neural architectures channel signal so that correlation is easy to discover. Great architectures also filter noise to help prevent overfitting.

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.

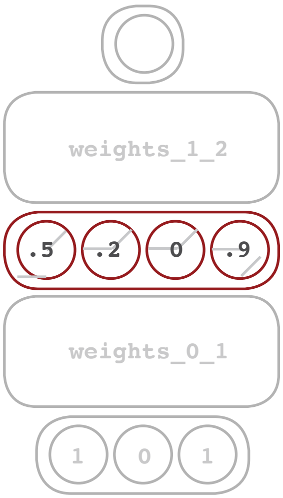
# Let's See This Network Predict

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

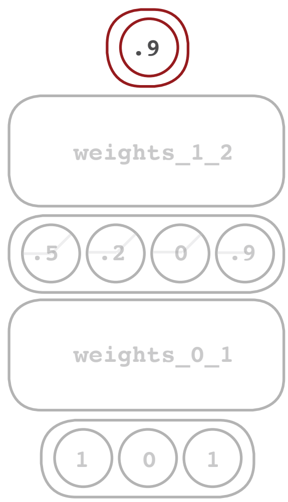
**1** A single datapoint from our "streetlight" dataset is selected. layer_0 is set to the correct values. (pictured right)




**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.



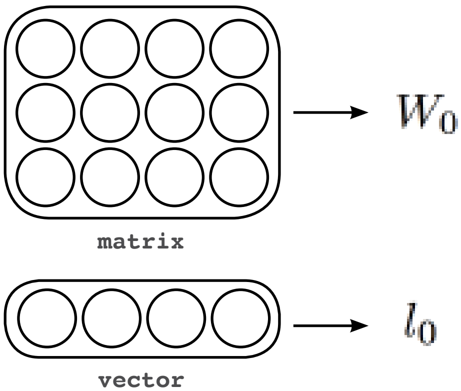

**Review: Vector-Matrix Multiplication**

Vector-Matrix multiplication performs multiple *weighted sums* of a vector. The matrix must have the same number of columns as the vector, so that each row in the matrix performs a unique weighted sum. Thus, if the matrix has 4 rows, 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

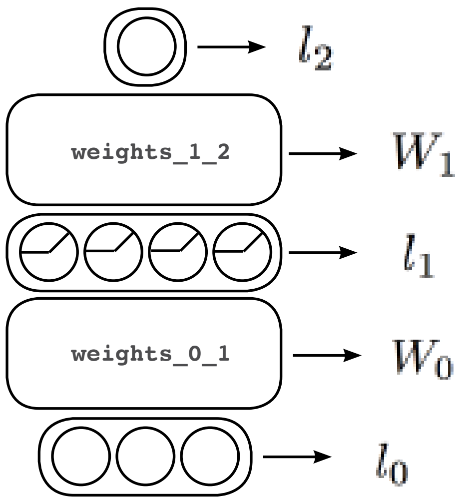
**All these pictures and detailed explanations are actually a simple piece of algebra.**

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.




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 nicer to make all of them "L"s and just number them than 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!



# 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.

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

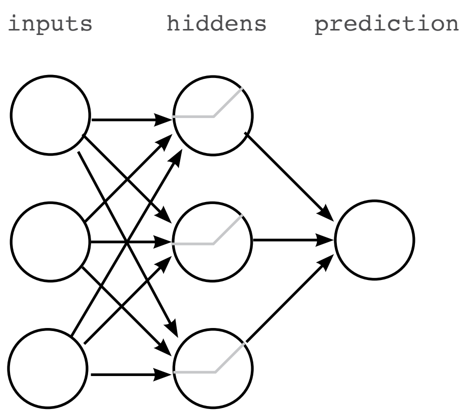
## Everything Side-by-Side

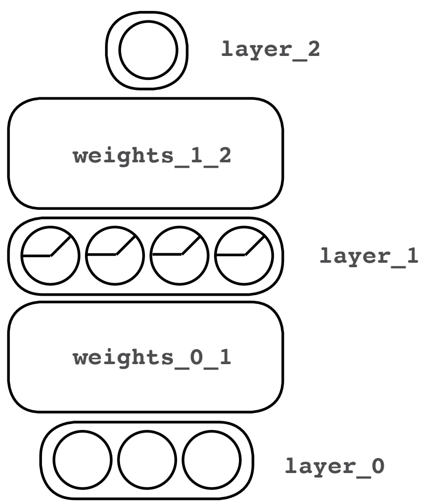
**Let's see our visualization, algebra formula, and python code in one place.**

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 novel 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 correaltion you're looking for. See you there!