The Mathematics of Learning

This article is meant to help myself and others understand the mathematics of backpropagation - a type of learning for neural networks - more deeply.

Background

What is machine learning?

One of the most unique and important aspects of our brains is our ability to learn. Over time, we are able to accumulate knowledge and modify our behavior based on our past experiences. This process is called learning, and it is what has allowed us to create all of the technology that we now rely on. Machine learning, which is a subfield of Artificial Intelligence, is the study of how to make computers learn in similar ways to humans.

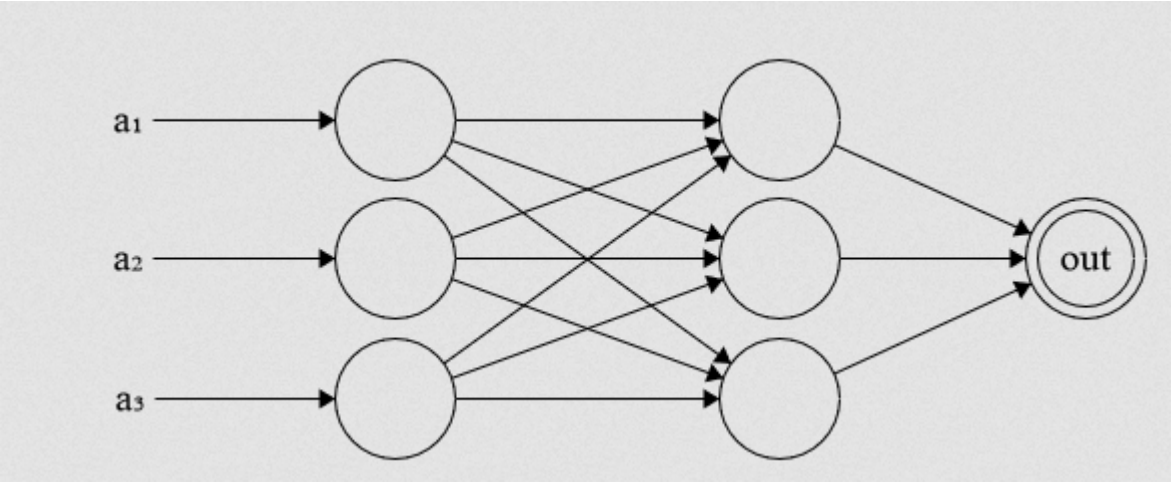
In traditional computer code, the output is determined by a human and does not change. If you run the same program a thousand times, it will give you the same result every time. It has no capability to learn, and if it makes a mistake a developer must manually modify the code.

Machine learning algorithms, by contrast, can change their outputs based on *data*. In other words, these models learn in a similar way to humans; they are given an input and an expected result, and attempt to recreate that result. Rather than a software engineer editing the model to make it better, the model is given a huge number of examples that it can model its behavior off of. Machine learning is relatively new in the field of AI, but it has shown remarkable results in a huge range of fields. With the advent of deep learning, there has been an explosion of applications in everything from facial recognition to autocomplete, and we are approaching human-like ability on tasks such as creative writing, language translation, and even driving a car.

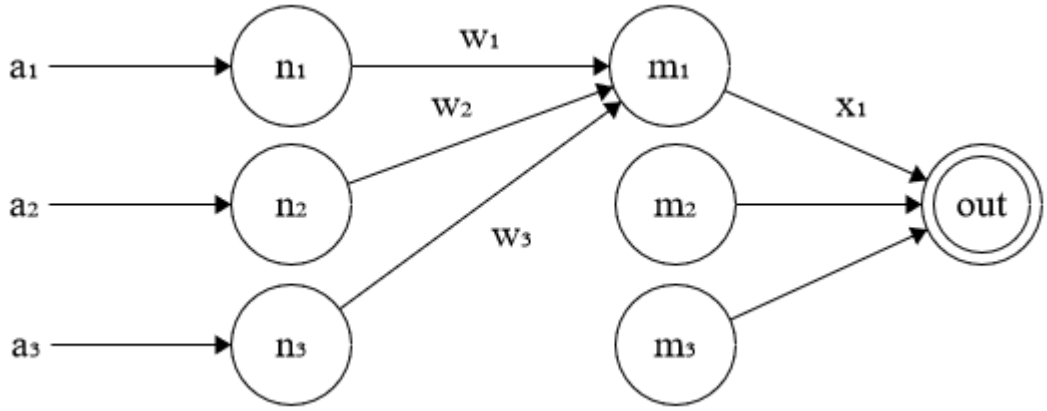
What are neural networks?

In each of our brains, there are billions of neurons with trillions of connections to each other. When we think or move or even when we are just sitting still, neurons send signals to each other. These signals make up our thoughts and actions, and travel between neurons through junctions called synapses. Our brain can modify these synapses by making them stronger or weaker, or by adding or removing them entirely. This modification of neural connections is what we call *learning*, and this is the concept that neural networks are based on. Neural networks are rough approximations of the networks in our brains, with "neurons" in layers, and each layer of neurons connected to each other.

Below is a visualization of a simple neural network with three layers:



In this image, the circles are the 'neurons', and the lines are the connections, or 'synapses'. In this network, each layer of neurons is connected to every neuron in the previous layer. To see how this structure allows for learning, let's prune some of the connections and add some labels:



In this network, data is input at a and follows the arrows until it reaches out. As you can see, m_1 takes the output of every neuron in the previous layer as input. The output of m_1 can be calculated with the formula $\sum_i^3 w_i * n_i + b_i$. The b term in this is called the **bias**, and is an additional parameter that the model will use to learn. This isn't the full story, as we still need a cost function and normalization function to create a fully-functional network.

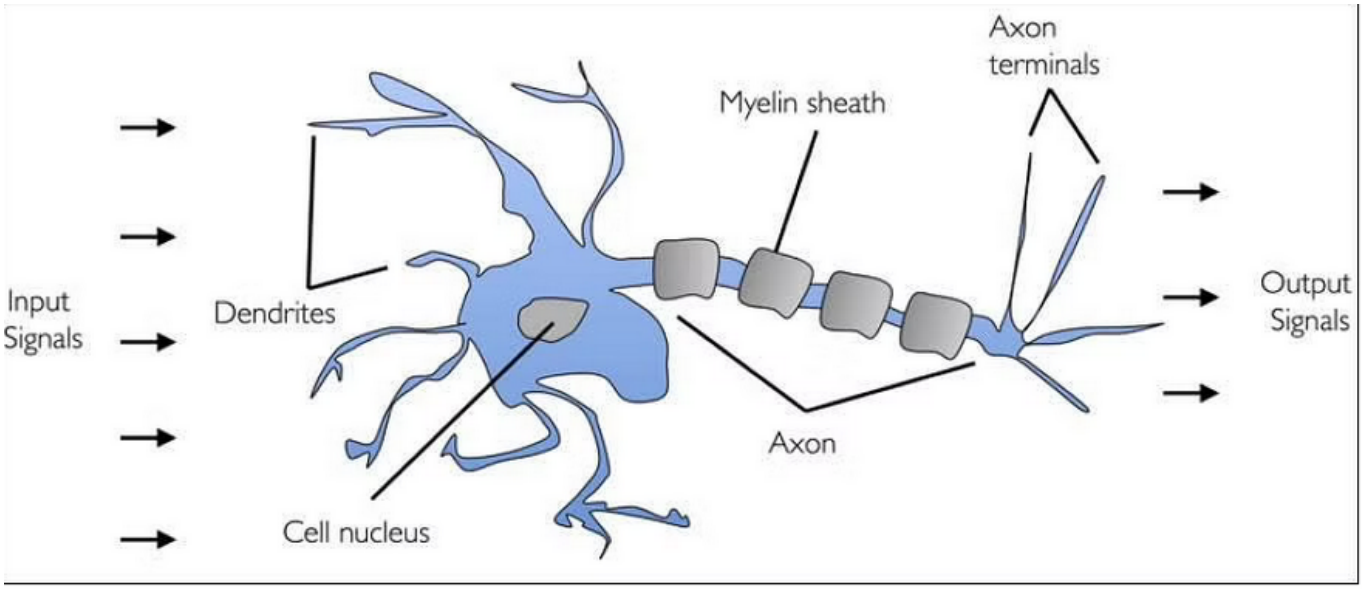
What is backpropagation?

So how can we give a model an example and expect it to learn? This process is called **backpropagation**, and is the process by which neural networks update their internal state based on input data.
Backpropagation, short for "backward propagation of error", was invented in 1970, but only became popular much more recently due to the emergence of deep learning. The algorithm uses the idea of **gradient descent** and the **chain rule** from calculus to allow errors to 'flow' backwards through neural networks.

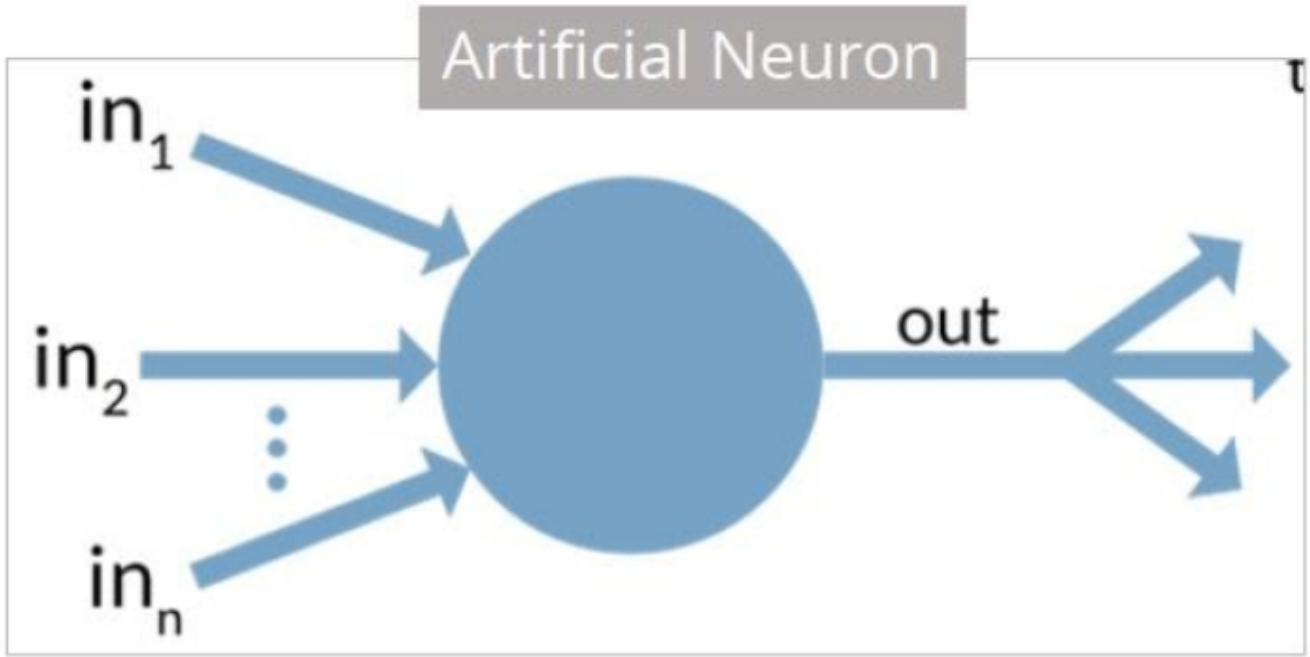
Neural Network Intuition

The building blocks of neural networks are **perceptrons**. Perceptrons are loosely based on biological neurons. In a neuron, there are many inputs, called dendrites, which are connected to other neurons. There is a single output, called an axon, which can connect to many neurons as well. When the neuron receives a strong enough signal from its dendrites, it fires a signal along its axon. A perceptron works very similarly, taking in many inputs and producing a single output that can be passed to many *other* perceptrons.

Neuron



Perceptron (artificial neuron)



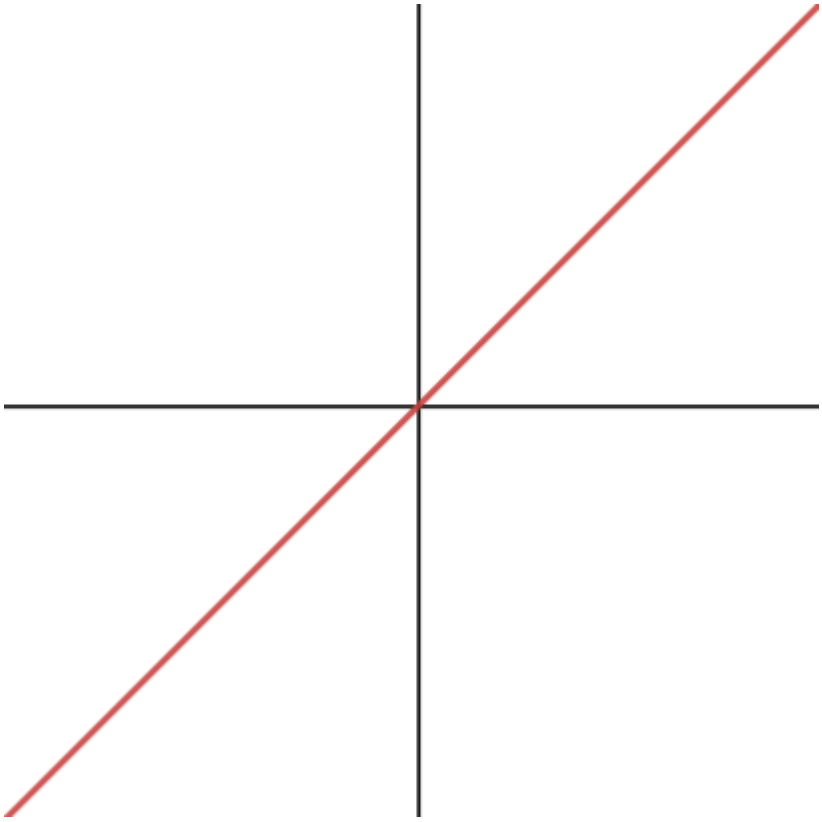
As neurons are the fundamental unit in the brain, perceptrons are the fundamental unit in neural networks.

When we learn, our brains generate mental models which approximate the things going on around us. While we learn to catch a ball, our brains slowly create a circuit that allows us to predict the ball's motion and catch the ball smoothly and efficiently. Our brains have created a mental model that *approximates* what is happening in the real world. We *predict* where the ball is going based on our visual input and move our hands into that position. In essence, our brains have created an approximation of the flight of the ball, which follows the equation:

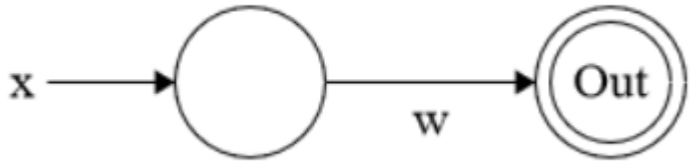
$$\overline{v} = \frac{\Delta x}{\Delta t}$$

Every single person has a circuit in their brains that approximates this equation with different levels of accuracy. Baseball players have an excellent approximation; they can quickly predict where a ball is going and are correct most of the time. Non-athletes are likely to have less accurate approximations; I can only catch balls that aren't thrown too fast or too high.

Just like how our brains learn approximations, neural networks do the same thing. Imagine you have a function $f(x) = x$. Its graph looks like this:



Say we want to teach a neural network to approximate this function. Let's start with the simplest possible neural network; a single perceptron.



This perceptron is a function of the input x and its weight parameter w, so we can write it as

$$f(x, w) = wx$$

To train this neuron, we can give it an input x and an expected output y and allow it to update its parameter w. (The internals of how it updates the parameters will come later)

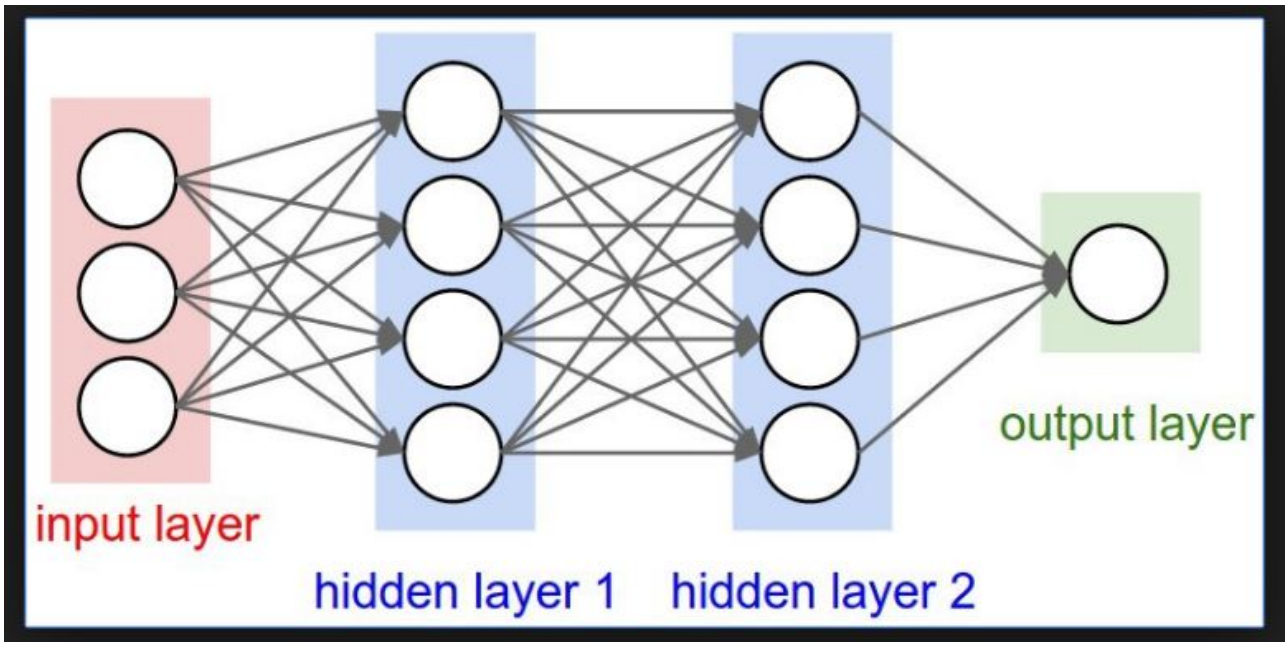
If we pick $x = 1$, then $y = 1$ and the correct value for w is 1. If we had a function $f(x) = 2x$ then $w = 2$.

The only problem with this single perceptron is that it's extremely limited. Adding more perceptrons to the network will allow us to approximate arbitrarily complicated *linear* systems, but we can't approximate something as simple as $f(x) = x^2$ with just this system.

To be able to handle higher-order functions, we need to add two things:

1. **Hidden layers**
   This is fairly simple. We add more layers of perceptrons to our network, which means that some layers are not directly accessible via the input or the output, hence *hidden*. This is where the term 'deep learning' comes from, and is what has made

modern machine learning so successful.



2. **Nonlinear activation functions**
   Adding hidden layers is great, but without the second step the output is *still* only a composition of linear functions. To be able to create anything but a straight line, we must introduce some "bend". We do this using *activation functions*. There are many types of these, but all of them take some input and produce some non-linear output (if you graph them, they don't produce a straight line).

But picking parameters for every perceptron requires a calculation of its inputs and outputs as well as the procedure for updating parameters. We also still don't know how to update the parameters for a network.

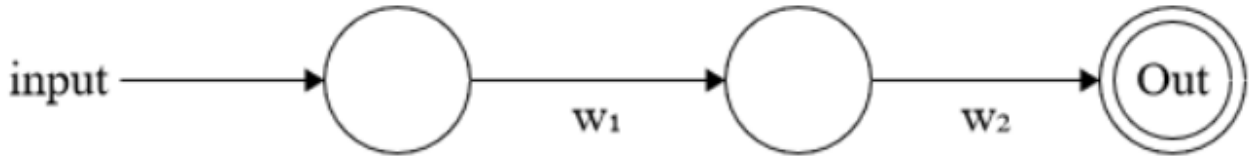
The Feedforward Algorithm

Introduction

The feedforward step of a neural network is how the network produces output from an input. In the single-perceptron example we saw above, the feedforward step was simply evaluating the function $f(x, w)$ to produce an output. In a larger neural network, the function is very similar, but includes more variables. A neural network with a single input and many parameters can be denoted as

$$f(x, \theta)$$

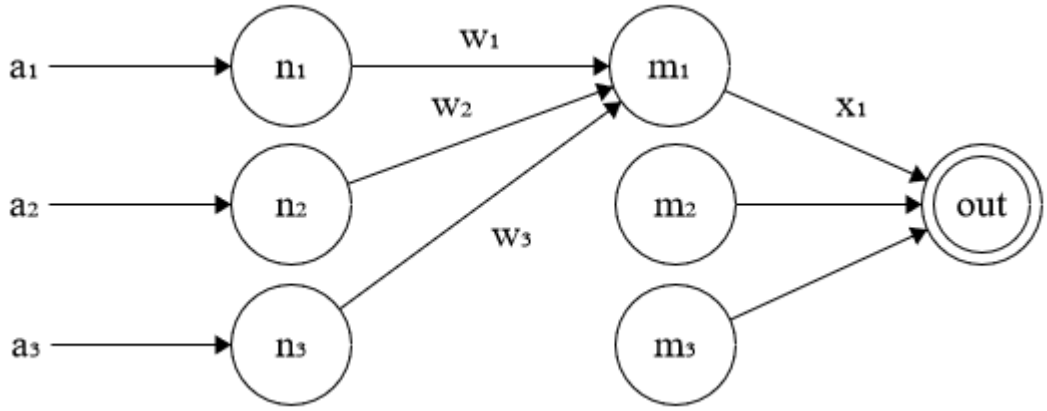
where θ is all of the parameters in the network. If we have a network that looks like this:



The feedforward function looks like this:

$$f(x, w_1, w_2) = w_1 * w_2 * x$$

Math



In the above model, an example input would be an array of numbers, for example [1,3,2]. Each node takes one of these numbers as their input and passes the input along to all of the nodes it is connected to. Since we have removed most of the connections, n_1, n_2, and n_3 only have a single output: the connection to m_1. As we saw before, the inputs are all multiplied by the *weight*, the value attached to the connections between nodes (w_1, w_2, and w_3 above); the *bias* is added on to the product of the input and the weight. To prevent this result from rapidly getting larger with successive multiplications, we apply a **normalization** function to the result, which "squashes" the result into a number on a fixed range. In this case we'll use the **sigmoid** function, which squashes the number onto the range of 0 to 1. The sigmoid function is below:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

So we now have the function to calculate the full output, or **activation** (denoted as a). We will write it in two steps for clarity, where z is the placeholder for the intermediate step.

$$z_m(a_i) = a_{ni} * w_i + b_m$$

And then

$$a_m = \sigma(z_m)$$

Finally, the output of the full network is:

$$\sum_m x_m * a_m$$

Which is the sum of all of the outputs of the m neurons multiplied by the weights x.

The Cost Function

Now that we have a way to calculate an output from an input, we need a way to calculate how far from the *correct* answer the output was. To do this, we define a **cost function** C which, given an input, returns the "distance" the predicted answer was from the correct answer. The prediction is denoted y and the correct answer (called the label) is denoted $\hat{y}$. There are many different cost functions, but for now we will use **Mean Squared Error (MSE)**

$$MSE(a, \hat{y}) = \frac{1}{2}(a - \hat{y})^2$$

> Note: All equations given are in their scalar versions rather than vector form. This is for notational and explanatory simplicity.

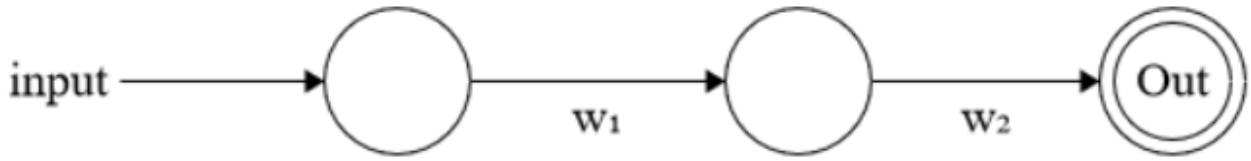
We now have a way to pass input through each layer of a neural network, and calculate the error of the network's output. This is the **feedforward** step of the learning process.

The Backpropagation Algorithm

Now that we have a way to calculate error, we can to use it to update the parameters (weights and biases) of the network. But we need one more thing; we need a *direction* to push the prediction to get it closer to the model. This is where calculus comes in.

Introduction

To update our network's output, we need to update the parameters (the weights and biases) of the network. We need a way to use the cost to tell each parameter how to change to move the output in the right direction. Let's go back to the simple two-perceptron model:



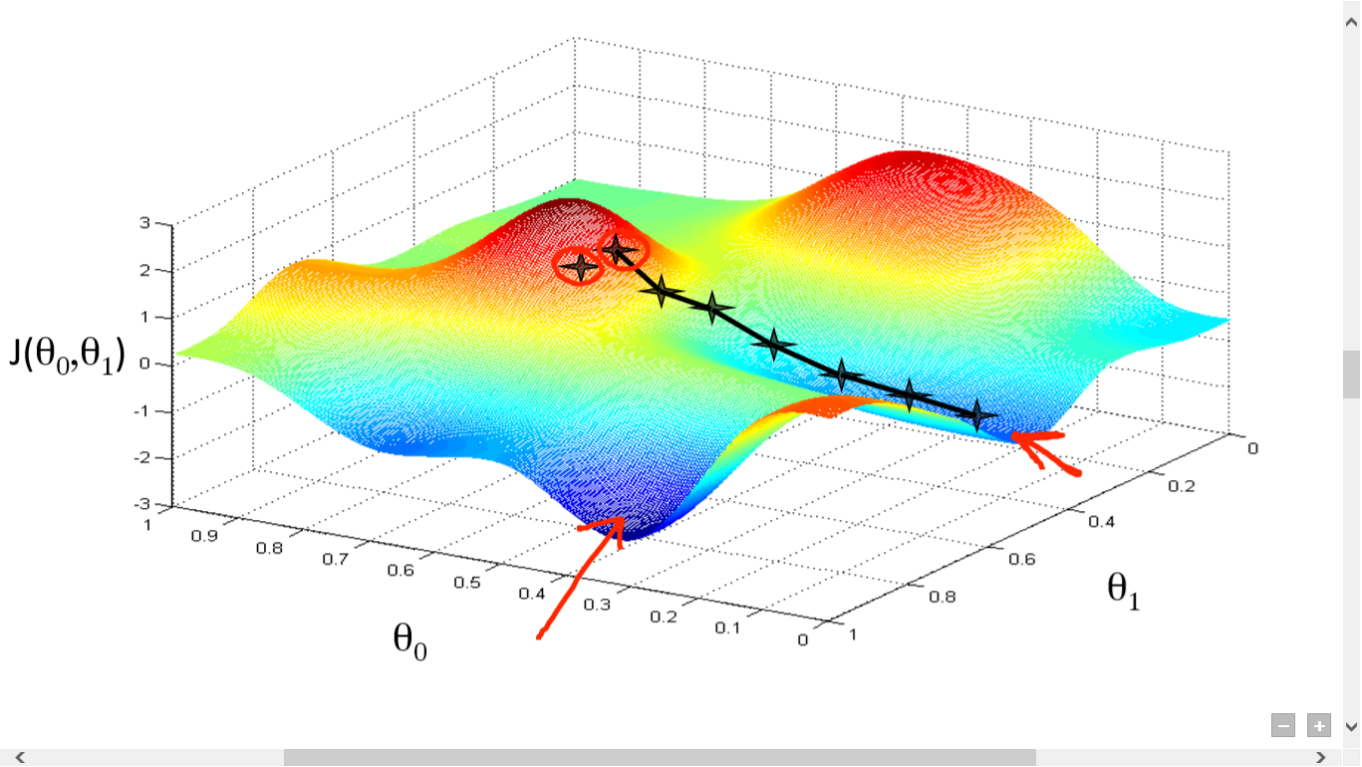
This model is a function of three variables:

$$f(x, \theta_1, \theta_2)$$

And the cost function is

$$C(x, \theta_1, \theta_2, y)$$

Let's provide an input and an expected output: $x = 1, y = 2$. Now C is only a function of the two parameters, and changes when we change those parameters. We can graph C as a function of our two parameters:



This graph tells us what coordinates of the parameters we can choose to make the cost the smallest. If we imagine a ball and place it on the graph, it will roll to the nearest low

point and stay there. This is how gradient descent works; **given an input and an expected output, how can we modify the parameters to minimize the cost function?**

Math

In physics, velocity (speed and direction) is calculated as the derivative of displacement in terms of time, $\frac{\delta d}{\delta t}$. It is basically saying "how much and in which direction does this thing move in a tiny period of time". To update our model parameters, we can use the same process. To update the weights of the last layer of our model, we need the derivative of the cost function with respect to the weights of the model (how much and in which direction does the cost function move with a small change in the weight?)

$$\frac{\delta C}{\delta w}$$

But C is a function of y and $\hat{y}$, so we need to use the **Chain Rule** to actually calculate this.

$$\frac{\delta C}{\delta w} = \frac{\delta C}{\delta a} * \frac{\delta a}{\delta z} * \frac{\delta z}{\delta w}$$

This is the derivative with respect to w of the composition of all of the functions we have seen. Keep in mind, the cost is calculated as $C(a(z(w,x)))$, so the derivative can be written as $C'(a(z(w,x))) * a'(z(w,x)) * z'(w,x) * w'$.

Now let's find the actual formula for this step-by-step.

1. $$\frac{\delta C}{\delta a} = a - \hat{y}$$

2. $$\frac{\delta a}{\delta z} = \sigma'(z)$$

3. $$\frac{\delta z}{\delta w} = a_{L-1}$$

    And the final function:

$$\frac{\delta C}{\delta w} = (a - \hat{y})\sigma'(z)a_{L-1}$$

> We now have a function that tells us how a 'nudge' in the weight parameter will change the cost. This value is what we use to update the weight parameters of the model.
> The bias derivation is very similar:

$$\frac{\delta C}{\delta b} = \frac{\delta C}{\delta a} * \frac{\delta a}{\delta z} * \frac{\delta z}{\delta b}$$

Which gives us

$$\frac{\delta C}{\delta w} = (a - \hat{y})\sigma'(z)$$

These values are calculated for every single weight and bias in the model, and the result is added to each parameter to provide a small 'push' down the 'hill' of the cost function. Over time, the parameters begin to cause the output to converge. The smaller the cost, the better the network has approximated the expected result.

Conclusion

Backpropagation is a complicated topic with lots of moving pieces, and there are likely points of confusion from this explanation. To supplement this article, I highly recommend 3Blue1Brown's Neural Networks explanation. It provides an excellent intuitive description of many of the topics discussed here, and has a ton of helpful animations.

It's easy to get lost in the weeds of neural network inner-workings, but the main concepts are these:
Neural networks are functions. They take in inputs and have parameters. They produce output based on the inputs and parameters. When 'learning', a neural network uses a cost function to tell it how to change it's parameters.

Definitions

Chain Rule: The chain rule of calculus is a rule by which we can calculate the derivatives of nested expressions. For example, the derivative of $3(x^2 + 1)$ can be done in three steps:

1. Calculate the derivative of the outside expression, and ignore the inside expression: $3(...) -> 3$
2. Calculate the derivative of the inside expression: $(x^2 + 1) -> 2x$
3. Multiply the two results together: $2x * 3 = 6x$

References

https://madebyevan.com/fsm/
https://brilliant.org/wiki/backpropagation/
https://people.idsia.ch/~juergen/who-invented-backpropagation-2014.html
https://www.desmos.com/calculator