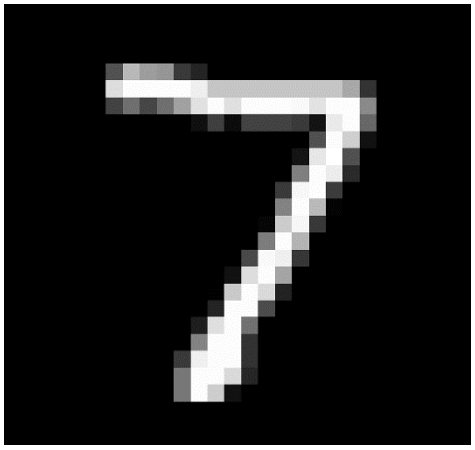
Exploring the Heart of Artificial Intelligence

With the release of ChatGPT in November 2022, many people have come to realise how powerful AI and deep learning can be, me included. The impressive awareness of the chatbot was quite surprising to me, as 'talking' to it was not dissimilar to talking to a living, breathing human. How can computers, with their binary inputs and outputs, replicate the human mind, learning and evolving to the point where a computer can hold entire conversations? The answer to this lies within machine learning, specifically *neural networks*.

As a simpler example of a problem solvable by neural networks, let's say we wanted to have a program recognise handwritten digits.

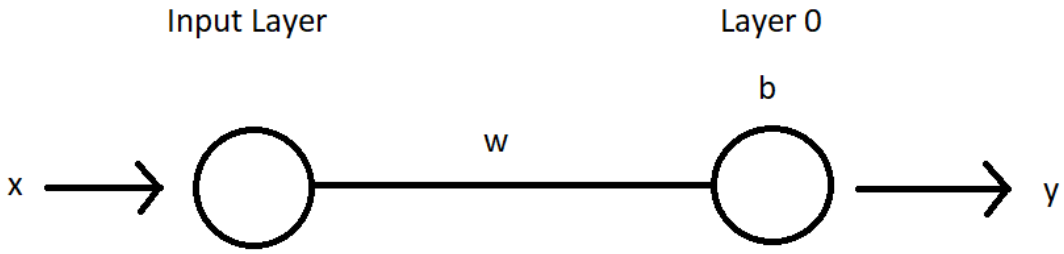


One way of thinking of this problem would be as a very complex function, something which takes in inputs and returns an output. The inputs to the function could be the brightness of each pixel, and our desired output is the digit drawn: in this case 7. Therefore, we need to find this function: some series of steps that transforms the brightness values to the digit shown in the image. Initially, this seems near impossible. How could we even figure out such a complicated function? This is where neural networks are used.

At a very simple level, a neural network is just a *function approximator*, making it a perfect solution for the problem. But how does it even work?

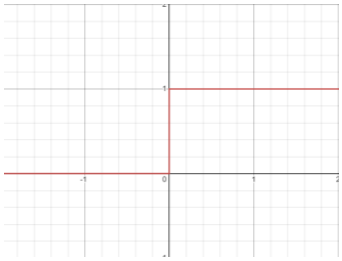
Introducing the neural network

To begin, this is what the simplest neural network would look like:

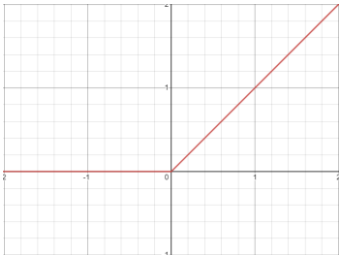


Each circle is referred to as a *node*, or neuron. Every node in one layer is connected to every node in the next layer, where a layer is a column of nodes. Nodes are connected by lines, referred to as *weights*, and each node has a *bias*: these weights and biases just represent numerical values. Also, it's worth pointing out that technically the inputs are not a layer, as they don't have weights or biases.

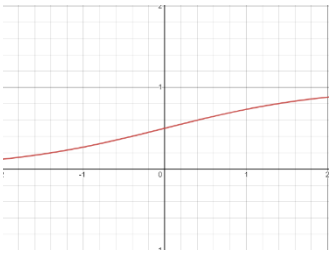
Despite looking unfamiliar, all this does is take in an input, x, multiply it by the weight, w, and add the bias, b, and return this new value as y. This leads to $y = wx + b$, an equation for a straight line. Now, by varying the weight and bias, we can change the resulting line. Unfortunately, this is far too simple for approximating anything useful as the resulting graph will only ever be linear. We can change this by passing the $wx + b$ (our *weighted inputs*, z) through an *activation function*. Activation functions are used to determine whether a node should be 'activated' or not: is the weighted input large enough to consider passing on to the next node? Examples of activation functions are:



Binary Step Function



ReLU (Rectified Linear Unit)

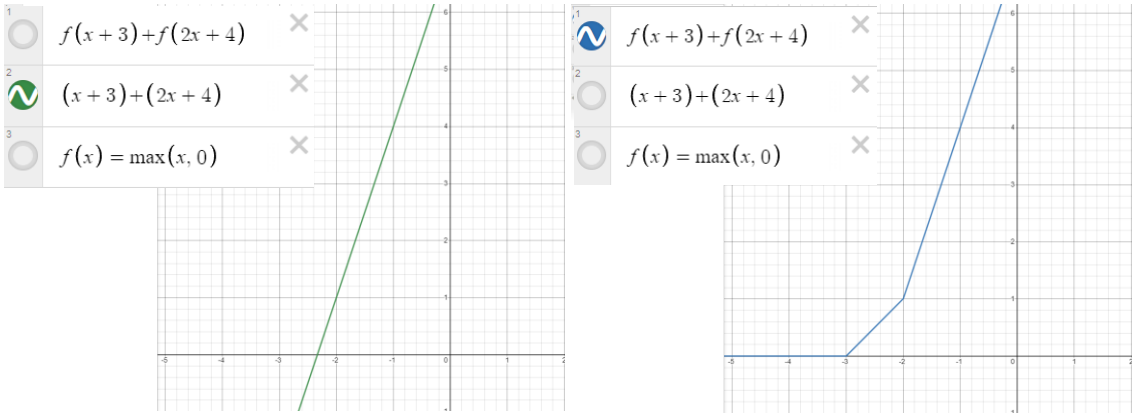


Sigmoid

You'll notice that the binary step function outputs 0 when the input is less than 0, and outputs 1 when the input is greater than 0 (e.g. when the node is activated). ReLU also outputs 0 when the input is less than 0, but leaves the input unchanged when the input is greater than 0. The sigmoid function is similar to the binary step function, with a smoother transition from 0 to 1 as the output. Despite there being multiple options for an activation function, only one will be used in all calculations across the whole neural network.

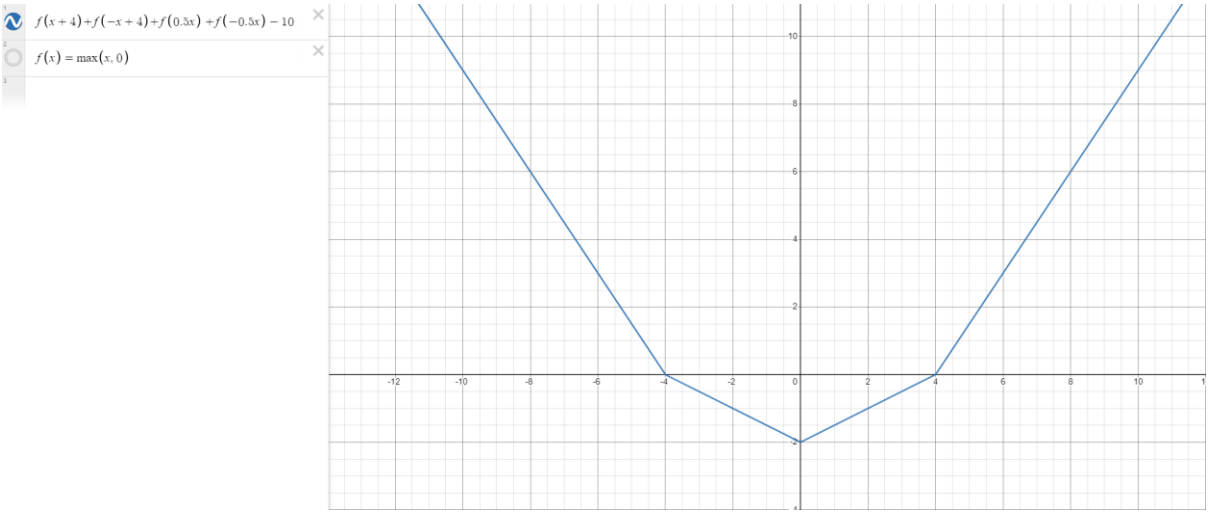
This is necessary as adding together two linear functions can only result in another linear function. However, adding together the same linear functions passed through an activation function can result in a non-linear function. Here is an example using ReLU:

$$ReLU\ function\colon\ f(x) = \max(x, 0)$$

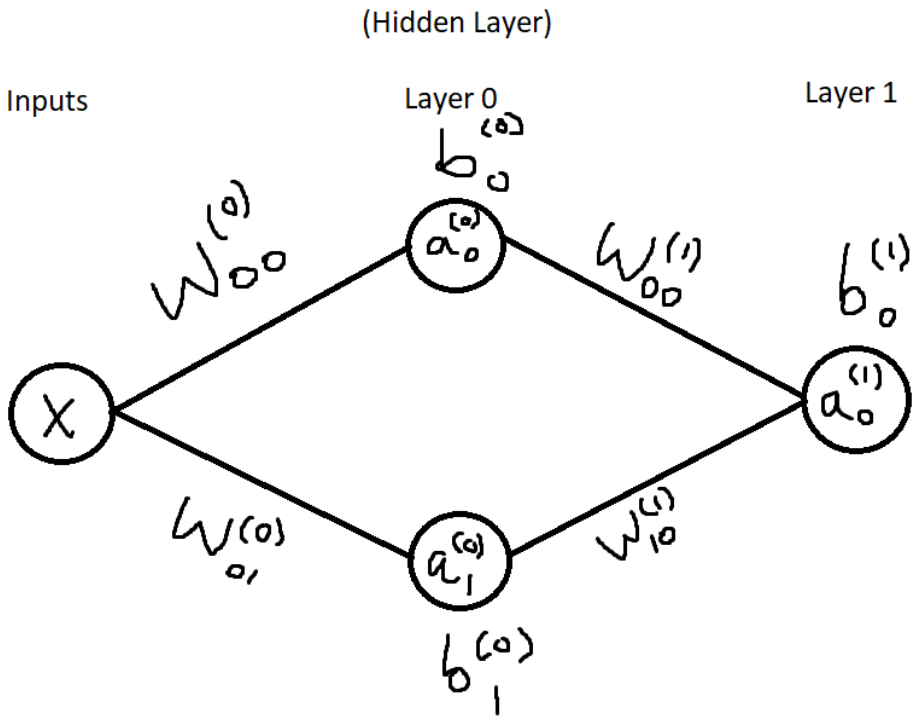


The first graph shows the shows the sum of two linear functions, while the second graph shows the sum of the two linear functions passed through the ReLU function

You can see how by adding together multiple linear functions passed through the ReLU function can 'build up' to make the shape of more complex graphs, such as (a very rough approximation of) a quadratic:

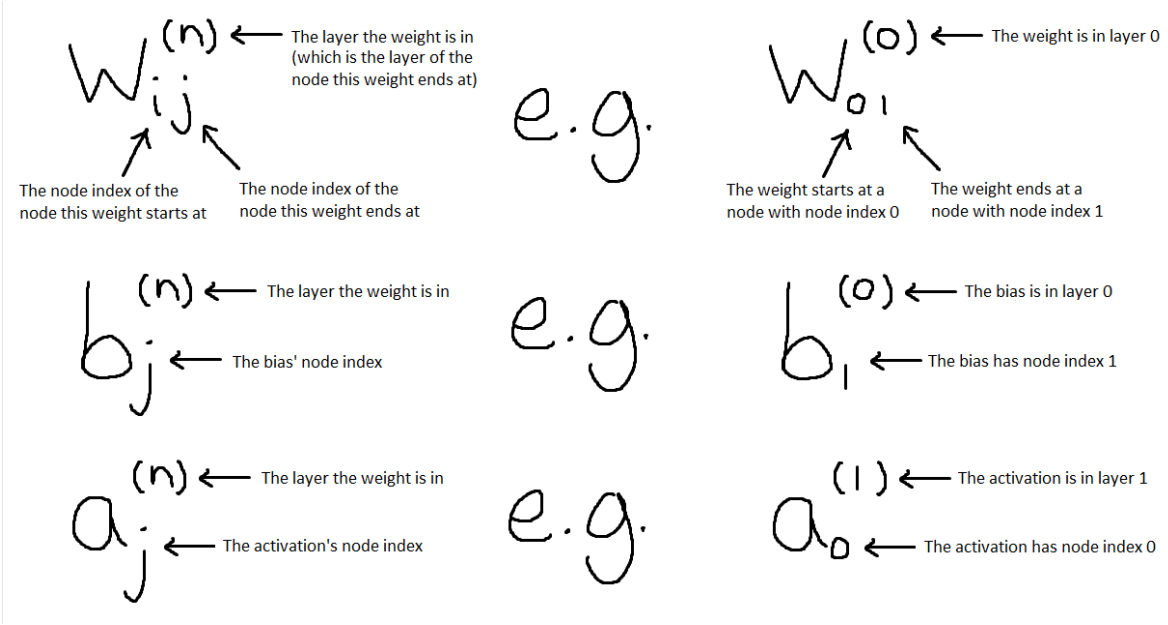


Therefore, using an activation function allows the neural network to approximate non-linear graphs. However, the network still needs to be more complex in order to make better approximations, and to do this we can add more weights and biases. Since we need to add more weights and biases, but can't change the size of the inputs or outputs, we can add another layer between the input and output layer. This is called a *hidden layer*. As example, we can add 2 nodes as a hidden layer to the previous neural network.



Note: 'a' from $a_0^{(0)}$, $a_1^{(0)}$, and $a_0^{(1)}$ in the nodes refers to the *activation* of that node, which is calculated by putting the sum of the weighted inputs into the activation function.

With these extra two nodes, the neural network already looks a lot more complicated, and therefore requires extra notation.



While this notation works, it would be much easier (and more accurate to how this is performed in a computer program) to use matrices (2 dimensional vectors, or arrays) to represent the weights, biases and activations of each layer. This is how that would look for any given layer n:

$$W^{(n)} = \begin{pmatrix} w^{(n)}_{00} & w^{(n)}_{10} & \cdots & w^{(n)}_{x0} \\ w^{(n)}_{01} & w^{(n)}_{11} & & \\ \vdots & & \ddots & \\ w^{(n)}_{0y} & & & w^{(n)}_{xy} \end{pmatrix} \quad e.g. \quad W^{(0)} = \begin{pmatrix} w^{(0)}_{00} \\ w^{(0)}_{10} \end{pmatrix}$$

$$B^{(n)} = \begin{pmatrix} b^{(n)}_{0} \\ b^{(n)}_{1} \\ \vdots \\ b^{(n)}_{x} \end{pmatrix} \quad e.g. \quad B^{(0)} = \begin{pmatrix} b^{(0)}_{0} \\ b^{(0)}_{1} \end{pmatrix}$$

$$A^{(n)} = \begin{pmatrix} a^{(n)}_{0} \\ a^{(n)}_{1} \\ \vdots \\ a^{(n)}_{x} \end{pmatrix} \quad e.g. \quad A^{(0)} = \begin{pmatrix} a^{(0)}_{0} \\ a^{(0)}_{1} \end{pmatrix}$$

Keep in mind that this is still basically the same notation as before, just in a more compact form. The same steps of multiplication, addition, and passing through the activation function will take place. Also, note that capital letters represent matrices, while lowercase refers to the individual weights, biases, weighted inputs, and activations.

Now we have a very compact way of calculating activations, where for any layer n:

$$A^{(n)} = f(W^{(n)} \cdot A^{(n-1)} + B^{(n)})$$

where f(x) is the activation function (the function is applied separately to each element of the resulting weighted input vector).

As an example of how the output is calculated, we can assign random values to each weight and bias:

$W^{(0)} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$, $B^{(0)} = \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}$, $W^{(1)} = (1 \quad 0.5)$, $B^{(0)} = (0)$,

And let the inputs, X, be (1). For this example I will use the ReLU function.

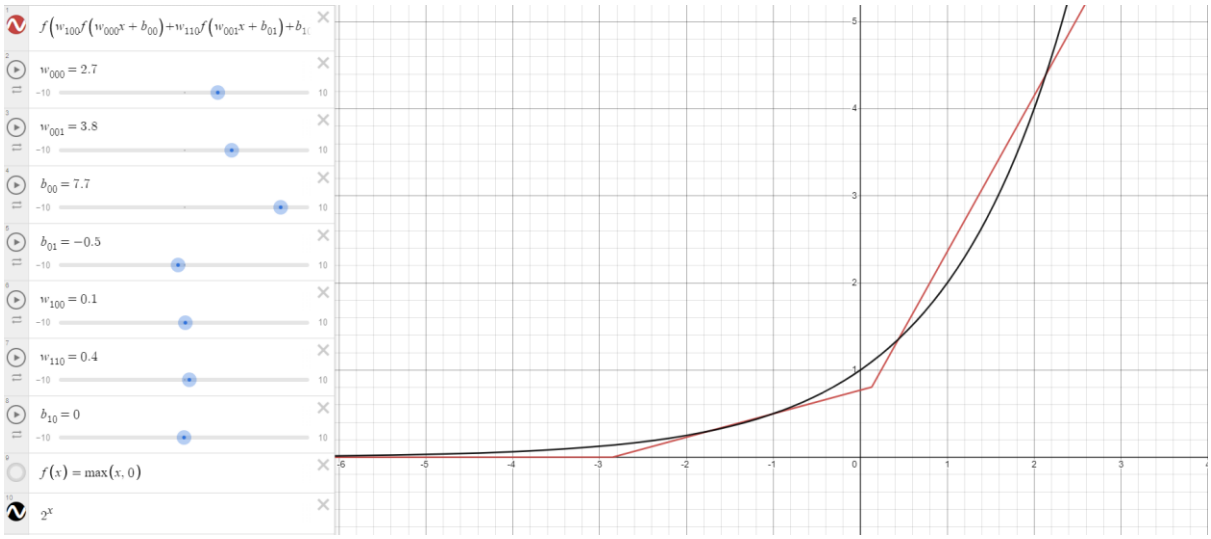
$$A^{(0)} = f\left(W^{(0)} \cdot X + B^{(0)}\right) = \begin{pmatrix} \max(0, 1 \cdot 1 + 0.5) \\ \max(0, -1 \cdot 1 + 1) \end{pmatrix} = \begin{pmatrix} 1.5 \\ 0 \end{pmatrix}$$

$$A^{(1)} = f\left(W^{(1)} \cdot A^{(0)} + B^{(1)}\right) = (\max(0, 1 \cdot 1.5 + 0 \cdot 0.5 + 0)) = (1.5)$$

Therefore the output would be 1.5. This function can also be shown visually with a graph:

And you can see that when x = 1, y = 1.5, so our calculations are correct.

By tweaking the values of the weights, we can actually make an attempt to approximate some functions. Here is an example of an approximation of an exponential function, 2^x:



The black curve shows $y = 2^x$, while the red shows the approximation

While this clearly isn't perfect, if we increased the number of nodes in the hidden layer or increased the number of hidden layers, we could find values for the weights and biases so that the neural

network very closely approximates the function for 2^x. Using matrices has also made the calculations very scalable: increasing the size of the hidden layers just increases the size of the matrices, and increasing the number of hidden layers just increases the number of times we need to iterate through the calculations for the activations in each layer.

So now we have a way of calculating how inputs are transformed into outputs within a neural network. However, a big question remains: how do we figure out the best values for the weights and biases?

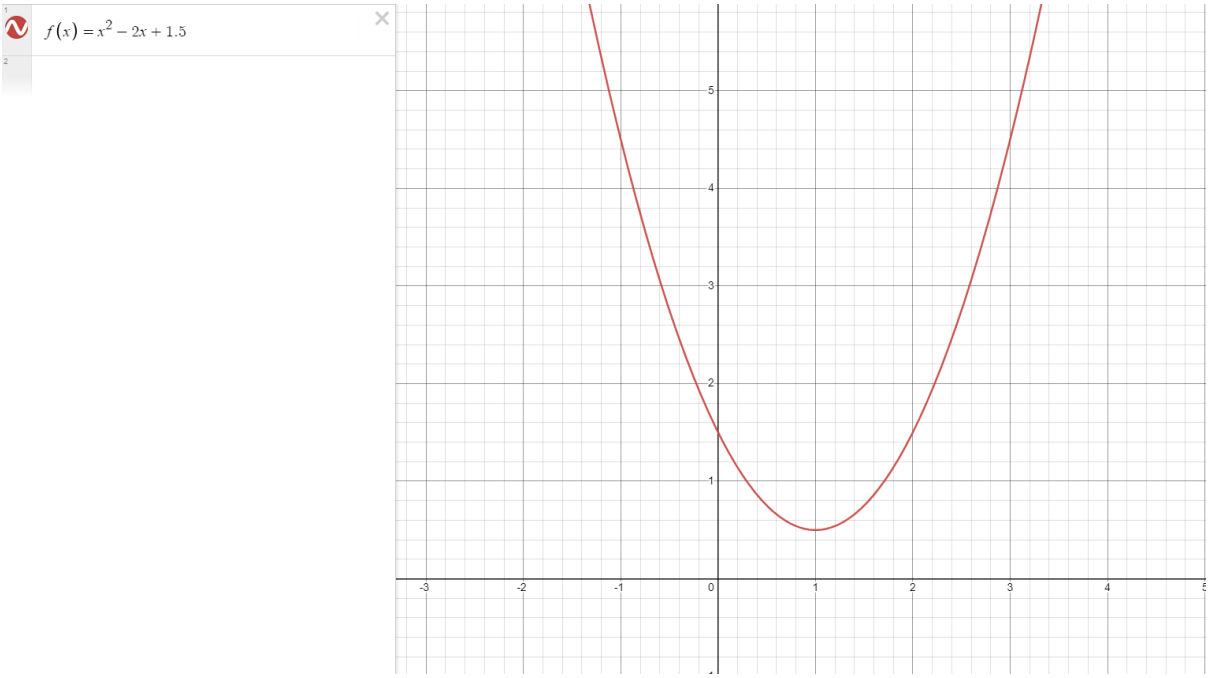
How it learns

First, we need a method to find how close the outputs are (with their random weights and biases) to the desired output. For example, when we inputted 1 into the neural network previously, the output was 1.5. However, the expected output may have been 0. Therefore, the error of this result is 1.5 – 0 = 1.5, although in neural networks, this error is squared (to make the result positive and highlight larger errors) to form the *cost function*:

$$C(\hat{y}, y) = (\hat{y} - y)^2$$

where ŷ is the output from the neural network (predicted output) and y is the expected output. Now, we need to figure out how each weight and bias affects the cost to be able to set the values so that the cost is as low as possible (therefore the neural network is most accurate).

We can think of the cost visually, even representing it using a graph:



In this example, we can see the cost function is lowest at around x = 1. The x axis here would represent one of the weights in the network, meaning that to have the smallest amount of error in our output, this weight should be around 1: this is our desired weight. This can be shown mathematically by finding where the gradient or slope of the graph is 0, hence we find the derivative (using the power rule) and set it to 0, giving us:

$$f(x) = x^2 - 2x + 1.5$$

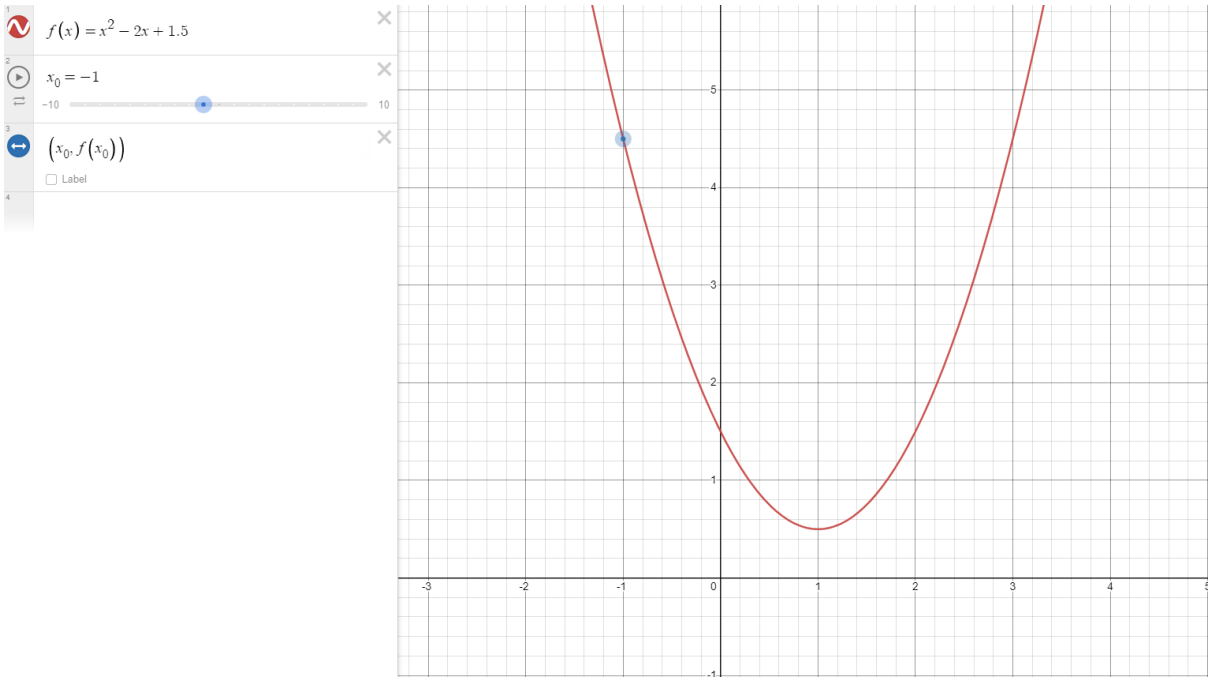
$$derivative\ of\ f(x) = f'(x) = 2x - 2$$

$f'(x) = 0 => 2x - 2 = 0$

$x = 1$

However, this method does not scale up well at all. Seeing as how multiple weights affect the output, this has the effect of bringing the graph into higher dimensions, making it extremely computationally inefficient. Instead, we use a method called *gradient descent*. This method is more efficient as it iteratively adjusts each weight and bias separately, allowing for simpler, scalable calculations.

Gradient descent begins by selecting random values for the weights and biases. If we focus on just one of the weights, we can show this on the previous graph:



The randomised value, x_0, of this weight is -1. Now, we know this value needs to be increased to get the optimal value for the weight, and because the cost (y value) is quite high, the x value should be increased by a somewhat large amount. Mathematically, these attributes (size and direction of the required change in the value of the weight) are shown by the gradient of the graph at x_0: a large gradient means the point is relatively far away from the minimum (you can see in the diagram that the gradient's magnitude increases as you get further from the minimum), and a negative gradient means the minimum cost is to the right of the point (as shown in the diagram), meaning if the point is to move towards the minimum we must subtract the gradient (to move the point in the positive direction).

We find the gradient as we did before, by taking the derivative of the function. You may wonder how this is more efficient than the previous method if we still need to find the gradient. The difference is that with this method, we only find the derivative with respect to one weight or bias at a time, making it scale up much better, as we won't end up with huge, multivariable equations to solve as we add more weights and biases. The reason we couldn't just take the derivative with respect to one weight or bias at a time previously was because (due to the weights and biases being so interconnected) changing one weight can affect how much another weight affects the cost. The iterative process of gradient descent counteracts this by always recalculating the gradient of the cost

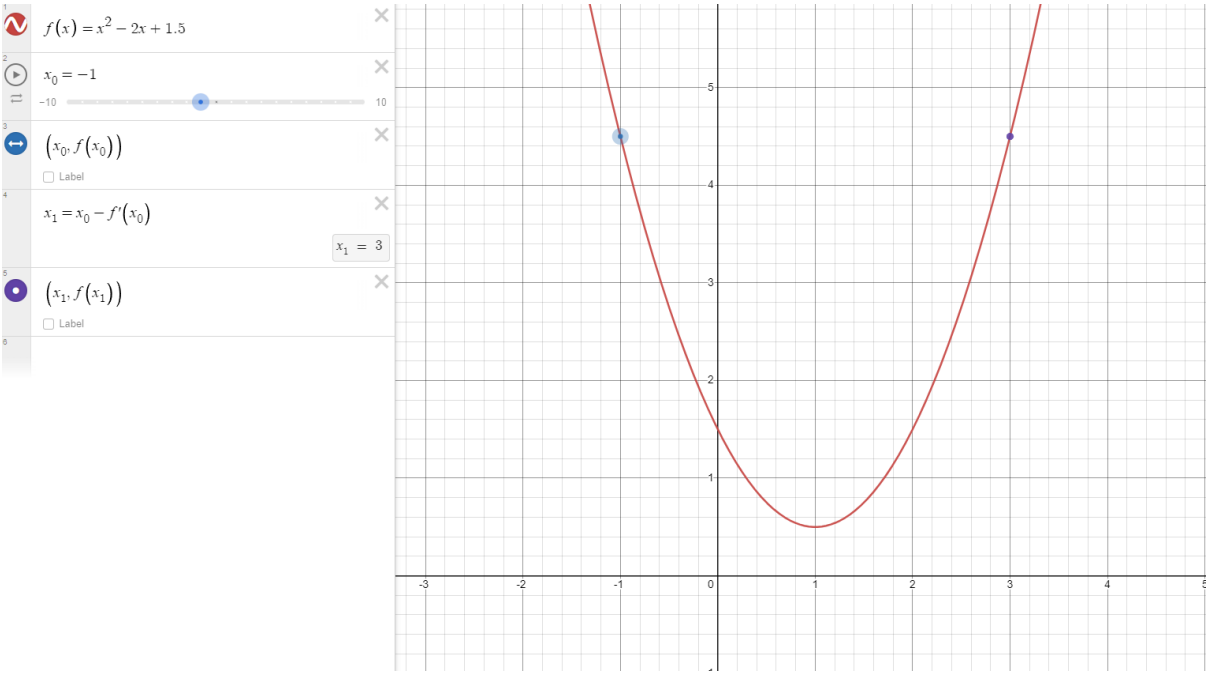
at the point, meaning even if a previously altered weight changed how the current weight affects the cost, this will be accounted for in the next iteration.

Now you understand why gradient descent is used, we can continue with the method. As previously mentioned, subtracting the gradient from the value of the weight will cause the weight to move closer to x = 1, which would minimise the cost. This can be shown mathematically:

$$x_0 = -1$$

$$x_1 = x_0 - f'(x_0) = -1 - (2x_0 - 2) = -1 - (2(-1) - 2) = 3$$

meaning the new weight after the first iteration will be $x_1 = 3$. Graphically, this looks like:



Unfortunately, it seems like the new point has overshot the minimum. If we carry on to the next iteration:

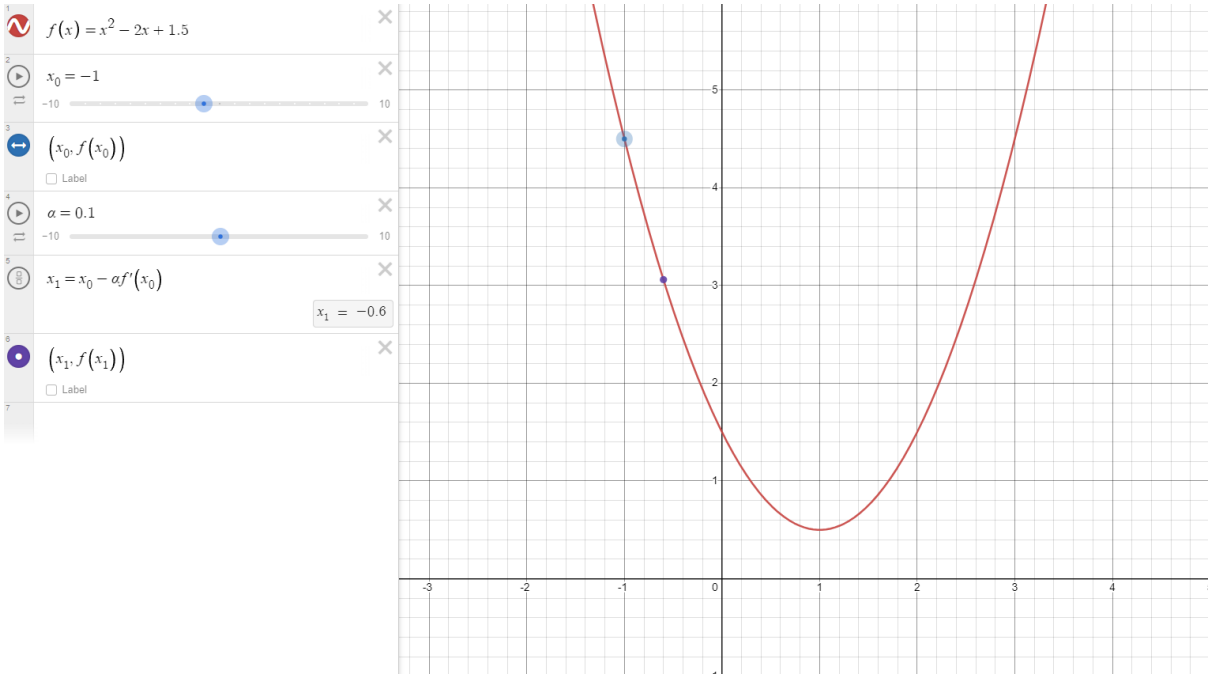
$$x_1 = 3$$

$$x_2 = x_1 - f'(x_1) = 3 - (2x_1 - 2) = 3 - (2(3) - 2) = -1$$

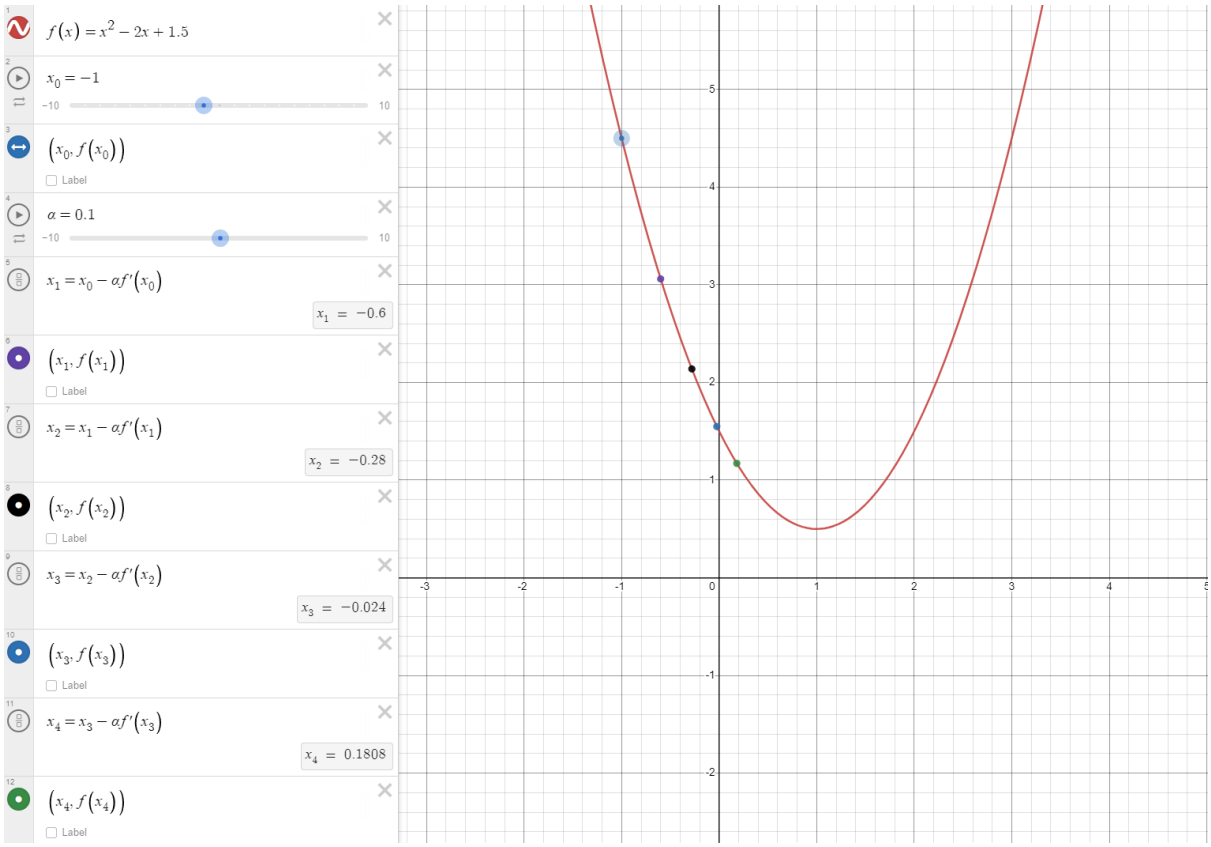
it becomes clear that the weight will never get closer to the minimum as it's stuck in a loop. This is due to the gradient of the graph at x = -1 being too large. We can remedy this by multiplying the gradient by some small constant before subtracting it from the weight. This constant is called the *learning rate*, α, as it determines how quickly the weight tends to the minimum. For this example, let α = 0.1. The new calculations are:

$$x_0 = -1$$

$$x_1 = x_0 - \alpha f'(x_0) = -1 - \alpha(2x_0 - 2) = -1 - 0.1(2(-1) - 2) = -0.6$$



Now it's clear that the new value of the weight has actually moved slightly closer to the minimum. As I mentioned before, this is an iterative process, so let's see how the point moves as we increase the number of iterations.



You can see that with more and more iterations, the weight approaches 1, the weight at which the cost is at its minimum. And this process can be repeated hundreds of times to be more and more accurate while barely taking any time for a computer to process, as the only calculations done are simple multiplications and additions. This is why a small learning rate wouldn't affect the outcome

too much, as after a thousand iterations the weight will be extremely close to 1 if the learning rate is 0.1 or 0.01. As long as the learning rate isn't absurdly small (or too large, as seen earlier with no learning rate, basically setting the learning rate to 1), gradient descent will work.

As a side note, writing out all the equations for x_0, x_1, x_2, etc, as shown in the last diagram can be tedious, so a nicer way of writing it can be found instead, simply by observing the pattern:

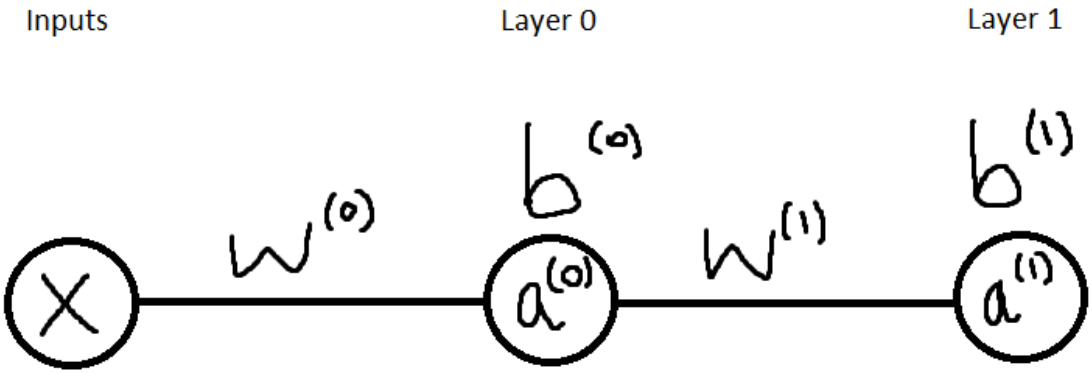
$$x_n = x_{n-1} - \alpha f'(x_{n-1})$$

Keep in mind that for each iteration, this process of subtracting the gradient is done for every weight and bias, so by the end of this, we will have a neural network with correctly tuned weights and biases to perform whatever task it was trained to do (e.g. identify handwitten digits).

However, we are still left with a problem. How can we calculate the derivative gradient of the cost as our neural network grows in size, and the calulations for the final predicted ouput contain more and more variables? Even with a small neural network, it won't be as simple as before with the quadratic graph, as the cost function isn't polynomial (a function containing multiple terms of x with differing powers) so we can't simply apply the power rule (as if you recall, it contains non-linear activation functions as well as multiple different variables).

Calculating the gradient

To begin with, we will be using a simple neural network with two layers.



We know that the cost is defined by the function:

$$C(\hat{y}, y) = (\hat{y} - y)^2$$

where $\hat{y}$ is the ouput from the neural network, which generally is $A^{(L)}$ (Note: L refers to the last layer in the neural network) and in this case is $a^{(1)}$. This is calculated by:

$$\hat{y} = A^{(L)} = f\left(Z^{(L)}\right) = f(W^{(L)} \cdot A^{(L-1)} + B^{(L)})$$

where $Z^{(L)}$ is the weighted inputs. For the neural network in the diagram, $\hat{y}$ is calculated by:

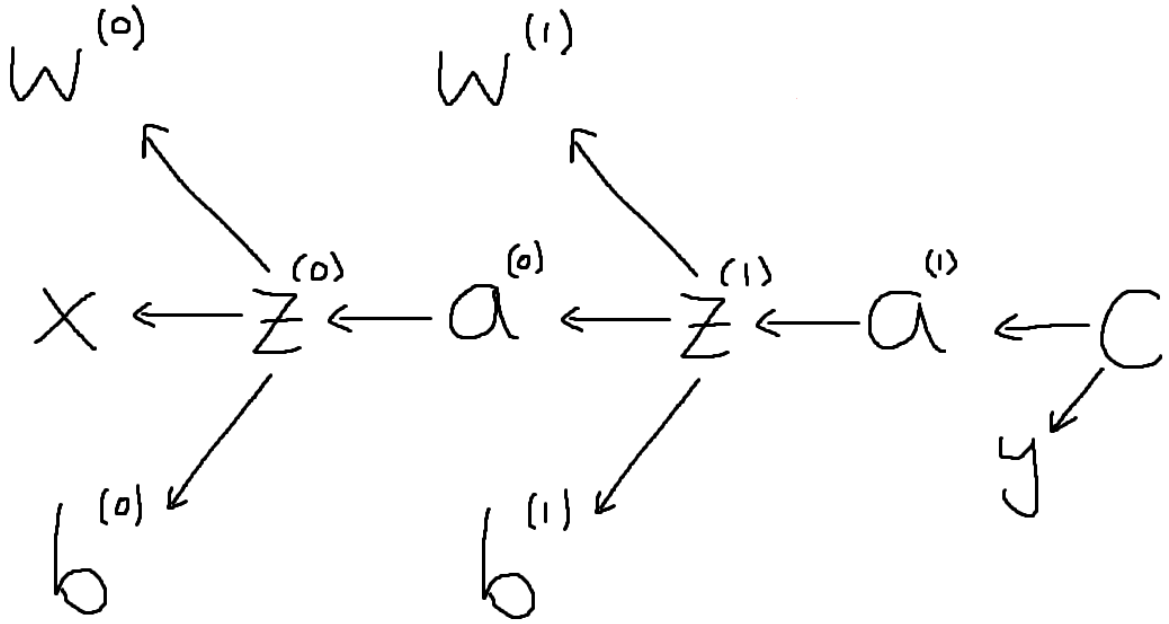
$$\hat{y} = a^{(1)} = f(z^{(1)}) = f(w^{(1)} \cdot a^{(0)} + b^{(1)})$$

and for this example I will use the sigmoid function as the activation function, so

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

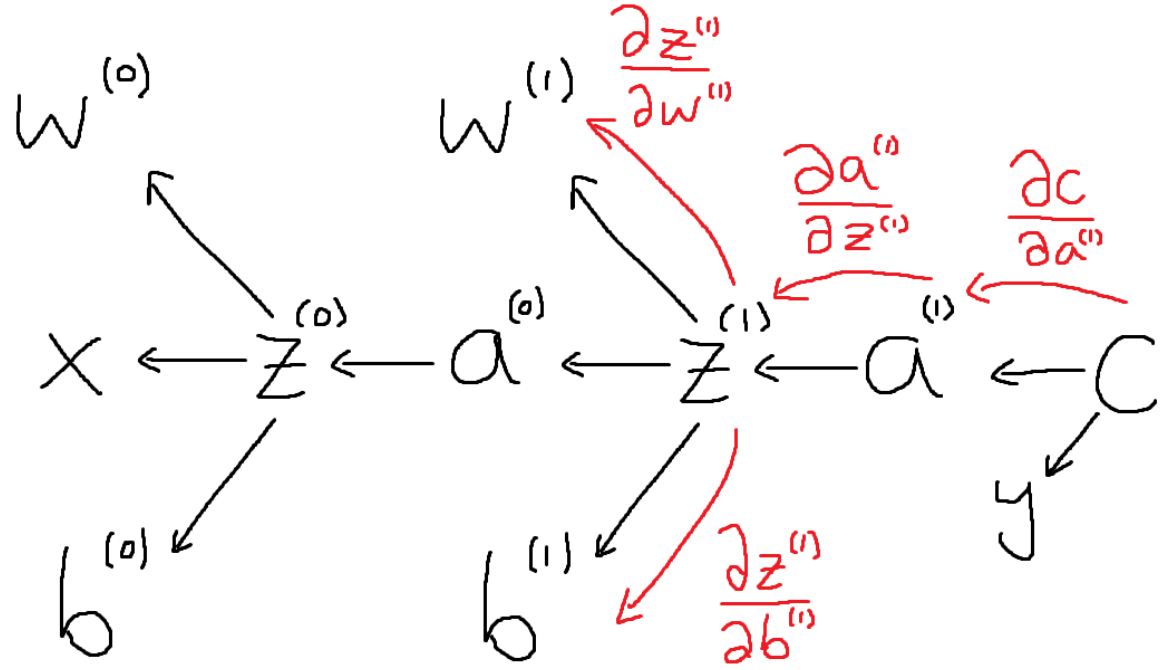
This is due to a useful property to do with its derivative which I will go into more depth about later.

First, let's draw a diagram to represent all the factors affecting the cost.



One thing to immediately note is that the expected output, y, is not connected to the other variables in any way and is also constant, so for the following calculations, we can just ignore it.

Now, if we wanted to find out the derivative of the cost with respect to $w^{(1)}$, we can see from the diagram that it doesn't directly affect the cost. Instead, $w^{(0)}$ affects $z^{(1)}$, which affects $a^{(1)}$, which affects the cost, c. So instead of directly finding the derivative of c with respect to $w^{(1)}$, we can find the derivative of c with respect to $a^{(1)}$, the derivative of $a^{(1)}$ with respect to $z^{(1)}$, and the derivative of $z^{(1)}$ with respect to $w^{(1)}$ to form a 'chain' of derivatives describing how $w^{(1)}$ indirectly influences c.



Note: If you are unfamiliar with the notation, $\frac{\partial c}{\partial a^{(1)}}$ represents the partial derivative of c with respect to $a_0^{(1)}$, and a partial derivative is just the derivative of a function containing multiple variables with respect to only one variable (so we treat other variables as constants).

If we wanted to use these terms to find the derivative of c with respect to $w^{(1)}$, according to the chain rule we just need to multiply the terms together.

$$\frac{\partial c}{\partial w^{(1)}} = \frac{\partial z^{(1)}}{\partial w^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}}$$

Similarly for the derivative of c with respect to $b_0^{(1)}$:

$$\frac{\partial c}{\partial b^{(1)}} = \frac{\partial z^{(1)}}{\partial b^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}}$$

This makes intuitive sense if you think of the terms as fractions (the derivatives are actually being expressed using Leibniz's notation, so aren't techically fractions but can be treated as such here), as you can simplify the expression on the right side of the equation to get the expression on the left.

Now we just need to find out what all these terms actually are, starting with $\frac{\partial c}{\partial a^{(1)}}$. Our equation for c in terms of $a^{(1)}$ is:

$$c = \left(a^{(1)} - y\right)^2 = {a^{(1)}}^2 - 2a^{(1)}y + y^2$$

Therefore, using the power rule:

$$\frac{\partial c}{\partial a^{(1)}} = 2a^{(1)} - 2y = 2(a^{(1)} - y)$$

Next up is to find $\frac{\partial a^{(1)}}{\partial z^{(1)}}$. Our equation for $a^{(1)}$ in terms of $z^{(1)}$ is:

$$a^{(1)} = f(z^{(1)})$$

Here is when that special property of sigmoids which I mentioned earlier becomes important, as for a sigmoid, f(x):

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

$$f'(x) = f(x) \cdot (1 - f(x))$$

Therefore:

$$\frac{\partial a^{(1)}}{\partial z^{(1)}} = f(z^{(1)}) \cdot (1 - f\left(z^{(1)}\right))$$

Finally, we need to find $\frac{\partial z^{(1)}}{\partial w^{(1)}}$ and $\frac{\partial z^{(1)}}{\partial b^{(1)}}$. Our equation for $z^{(1)}$ in terms of $w^{(1)}$ is:

$$z^{(1)} = w^{(1)} \cdot a^{(0)} + b^{(1)}$$

Therefore, again using the power rule:

$$\frac{\partial z^{(1)}}{\partial w^{(1)}} = a^{(0)}$$

$$\frac{\partial z^{(1)}}{\partial b^{(1)}} = 1$$

At last, we can form an equation for the derivative of the cost:

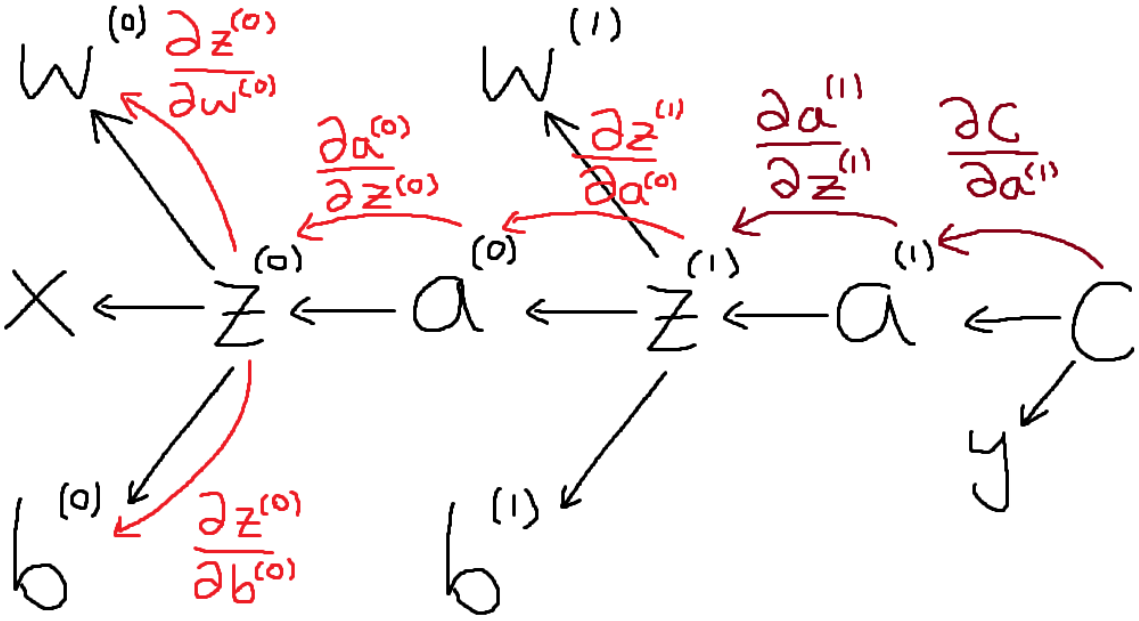
$$\frac{\partial c}{\partial w^{(1)}} = a^{(0)} \cdot f(z^{(1)}) \cdot (1 - f(z^{(1)})) \cdot 2(a^{(1)} - y)$$

$$\frac{\partial c}{\partial b^{(1)}} = f(z^{(1)}) \cdot (1 - f(z^{(1)})) \cdot 2(a^{(1)} - y)$$

This means when adjusting $w^{(1)}$ during the process of gradient descent, the actual calculation for the new $w^{(1)}$ performed will be (not including the learning rate mutliplier):

$$w^{(1)} - a^{(0)} \cdot f(z^{(1)}) \cdot (1 - f(z^{(1)})) \cdot 2(a^{(1)} - y)$$

So, we can now find the derivative of the cost with respect to $w^{(1)}$ and $b^{(1)}$ (or more generally, $w^{(L)}$ and $b^{(L)}$). But what about the derivative of the cost with respect to $w^{(0)}$ and $b^{(0)}$? Fortunately, finding these values will contain very similar calculations to what we've just done.



Now to find the derivatives, we multiply the terms together as we did before.

$$\frac{\partial c}{\partial w^{(1)}} = \frac{\partial z^{(0)}}{\partial w^{(0)}} \times \frac{\partial a^{(0)}}{\partial z^{(0)}} \times \frac{\partial z^{(1)}}{\partial a^{(0)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}}$$

$$\frac{\partial c}{\partial b^{(1)}} = \frac{\partial z^{(0)}}{\partial b^{(0)}} \times \frac{\partial a^{(0)}}{\partial z^{(0)}} \times \frac{\partial z^{(1)}}{\partial a^{(0)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}}$$

One thing you'll notice is that the two terms on the right (coloured in dark red), $\frac{\partial a^{(1)}}{\partial z^{(1)}}$ and $\frac{\partial c}{\partial a^{(1)}}$, are the same used in the previous equations, and this makes sense when looking at the diagram, as the 'path' to $w^{(1)}$ and $w^{(0)}$ from c both go c -> $a^{(1)}$ -> $z^{(1)}$. Another important observation is that the two terms on the left, $\frac{\partial z^{(0)}}{\partial w^{(0)}}$ or $\frac{\partial z^{(0)}}{\partial b^{(0)}}$ and $\frac{\partial a^{(0)}}{\partial z^{(0)}}$, are very similar to terms used in the previous equations, except with the layer index decreased by 1 (although keep in mind there is no $a^{(-1)}$ so this will

become x in the equation). The only actually new term is $\frac{\partial z^{(1)}}{\partial a^{(0)}}$, so let's calculate that now. The equation for $z^{(1)}$ in terms of $a^{(0)}$ is:

$$z^{(1)} = w^{(1)} \cdot a^{(0)} + b^{(1)}$$

Therefore:

$$\frac{\partial z^{(1)}}{\partial a^{(0)}} = w^{(1)}$$

And by using this new value, as well as the values we previously worked out for the other familiar terms, we get:

$$\frac{\partial c}{\partial w^{(0)}} = x \cdot f(z^{(0)}) \cdot (1 - f(z^{(0)})) \cdot w^{(1)} \cdot f(z^{(1)}) \cdot (1 - f(z^{(1)})) \cdot 2(a^{(1)} - y)$$

$$\frac{\partial c}{\partial b^{(0)}} = f(z^{(0)}) \cdot (1 - f(z^{(0)})) \cdot w^{(1)} \cdot f(z^{(1)}) \cdot (1 - f(z^{(1)})) \cdot 2(a^{(1)} - y)$$

These equations may look daunting, but if you look back at our equations for $\frac{\partial c}{\partial w^{(1)}}$ and $\frac{\partial c}{\partial b^{(1)}}$, most of the terms are the same or similar. However, if our neural network had more layers, you can imagine these equations would grow much larger as the path to get to the desired weight or bias becomes longer. Instead, we can use the fact mentioned before, about how a section of this path (c -> $a^{(1)}$ -> $z^{(1)}$), was repeated, leading to the $\frac{\partial a^{(1)}}{\partial z^{(1)}}$ and $\frac{\partial c}{\partial a^{(1)}}$ terms appearing again.

To avoid recalculating this section of the equation, $\frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}}$, we denote it with a variable, ***v***. We'll use this to save a lot of time when it comes calculating the derivative of the cost with respect to the weights and biases in earlier layers. Since all the repeated terms come from Layer 1, $L^{(1)}$, we can be more specific and call it $v^{(1)}$, where:

$$v^{(1)} = \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}} = f(z^{(1)}) \cdot (1 - f(z^{(1)})) \cdot 2(a^{(1)} - y)$$

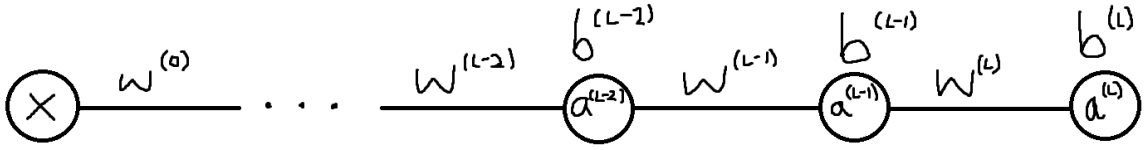
Therefore we can simplify the equations to:

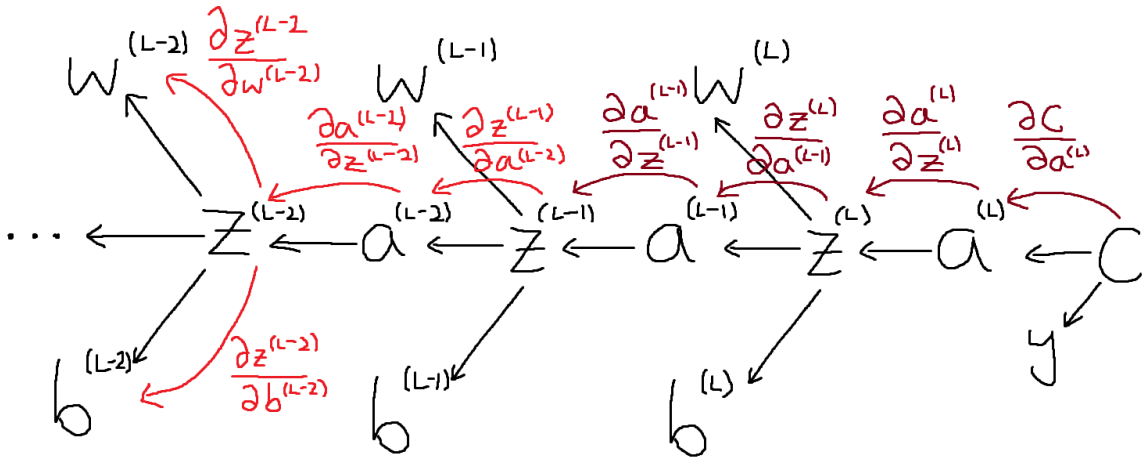
$$\frac{\partial c}{\partial w^{(0)}} = x \cdot f(z^{(0)}) \cdot (1 - f(z^{(0)})) \cdot w^{(1)} \cdot v^{(1)}$$

$$\frac{\partial c}{\partial b^{(0)}} = f(z^{(0)}) \cdot (1 - f(z^{(0)})) \cdot w^{(1)} \cdot v^{(1)}$$

Because using v allows these calculations to be a lot more efficient (as we don't need to recalculate most of the terms), finding the derivatives of the weights and biases is done backwards, starting with the weights and biases at the end of the neural network and working towards the beginning (as we've been doing). This is the process of *backpropagation*.

And you can see if we extended our neural network to one with L + 1 layers (going up to layer L as indexing starts at 0), we could use a new value for $v^{(L-1)}$ to make finding new derivatives easier:





Obviously we could multiply all the terms together, but that would be tedious and inefficient to recalculate all the terms. Instead, take note of how we already know the 4 rightmost terms (as at this point we would have calculated the weights and biases for layers L and L – 1 due to us working backwards), and set $v^{(L-1)}$ equal to them.

$$v^{(L-1)} = \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \times \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \times \frac{\partial a^{(L)}}{\partial z^{(L)}} \times \frac{\partial c}{\partial a^{(L)}}$$

Therefore:

$$\frac{\partial c}{\partial w^{(L-2)}} = \frac{\partial z^{(L-2)}}{\partial w^{(L-2)}} \times \frac{\partial a^{(L-2)}}{\partial z^{(L-2)}} \times \frac{\partial z^{(L-1)}}{\partial a^{(L-2)}} \times v^{(L-1)}$$

$$\frac{\partial c}{\partial w^{(L-2)}} = a^{(L-3)} \cdot f(z^{(L-2)}) \cdot (1 - f(z^{(L-2)})) \cdot w^{(L-1)} \cdot v^{(L-1)}$$

$$\frac{\partial c}{\partial b^{(L-2)}} = \frac{\partial z^{(L-2)}}{\partial b^{(L-2)}} \times \frac{\partial a^{(L-2)}}{\partial z^{(L-2)}} \times \frac{\partial z^{(L-1)}}{\partial a^{(L-2)}} \times v^{(L-1)}$$

$$\frac{\partial c}{\partial b^{(L-2)}} = f(z^{(L-2)}) \cdot (1 - f(z^{(L-2)})) \cdot w^{(L-1)} \cdot v^{(L-1)}$$

By either observing the pattern of values of v, or looking at the diagram, you may have noticed that $v^{(L-2)}$ will just be the last 3 terms of the equation for $\frac{\partial c}{\partial w^{(L-2)}}$ and $\frac{\partial c}{\partial b^{(L-2)}}$:

$$v^{(L-2)} = \frac{\partial a^{(L-2)}}{\partial z^{(L-2)}} \times \frac{\partial z^{(L-1)}}{\partial a^{(L-2)}} \times v^{(L-1)}$$

Or more generally:

$$v^{(n)} = \frac{\partial \mathrm{a}^{(n)}}{\partial \mathrm{z}^{(n)}} \times \frac{\partial \mathrm{z}^{(n+1)}}{\partial \mathrm{a}^{(n)}} \times v^{(n+1)}$$

Note: this only works for n < L. For $v^{(L)}$, the equation is:

$$v^{(L)} = \frac{\partial \mathrm{a}^{(L)}}{\partial \mathrm{z}^{(L)}} \times \frac{\partial \mathrm{c}}{\partial \mathrm{a}^{(L)}}$$

Now we can even write the general formula for the derivative of the cost with respect to a weight or bias in a hidden layer:

$$\frac{\partial c}{\partial \mathrm{w}^{(n)}} = \frac{\partial \mathrm{z}^{(n)}}{\partial \mathrm{w}^{(n)}} \times \frac{\partial \mathrm{a}^{(n)}}{\partial \mathrm{z}^{(n)}} \times \frac{\partial \mathrm{z}^{(n+1)}}{\partial \mathrm{a}^{(n)}} \times v^{(n+1)}$$

$$\frac{\partial c}{\partial \mathrm{w}^{(n)}} = a^{(n-1)} \cdot f(z^{(n)}) \cdot (1 - f(z^{(n)})) \cdot w^{(n+1)} \cdot v^{(n+1)}$$

$$\frac{\partial c}{\partial \mathrm{b}^{(n)}} = \frac{\partial \mathrm{z}^{(n)}}{\partial \mathrm{b}^{(n)}} \times \frac{\partial \mathrm{a}^{(n)}}{\partial \mathrm{z}^{(n)}} \times \frac{\partial \mathrm{z}^{(n+1)}}{\partial \mathrm{a}^{(n)}} \times v^{(n+1)}$$

$$\frac{\partial c}{\partial \mathrm{b}^{(n)}} = f(z^{(n)}) \cdot (1 - f(z^{(n)})) \cdot w^{(n+1)} \cdot v^{(n+1)}$$

Once again, this only works for n < L. For $v^{(L)}$, the equation is:

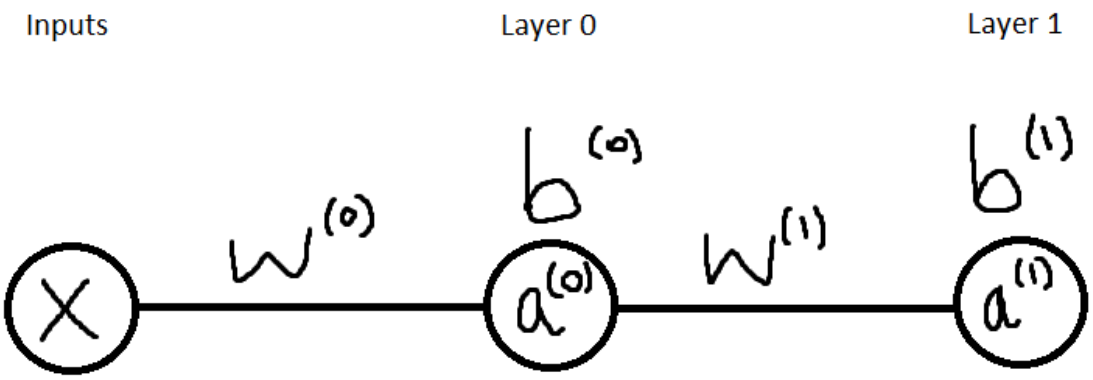
$$\frac{\partial c}{\partial \mathrm{w}^{(L)}} = \frac{\partial \mathrm{z}^{(L)}}{\partial \mathrm{w}^{(L)}} \times \frac{\partial \mathrm{a}^{(L)}}{\partial \mathrm{z}^{(L)}} \times \frac{\partial \mathrm{c}}{\partial \mathrm{a}^{(L)}}$$

$$\frac{\partial c}{\partial \mathrm{w}^{(L)}} = a^{(L-1)} \cdot f(z^{(L)}) \cdot (1 - f(z^{(L)})) \cdot 2(a^{(L)} - y)$$

$$\frac{\partial c}{\partial \mathrm{b}^{(L)}} = \frac{\partial \mathrm{z}^{(L)}}{\partial \mathrm{b}^{(L)}} \times \frac{\partial \mathrm{a}^{(L)}}{\partial \mathrm{z}^{(L)}} \times \frac{\partial \mathrm{c}}{\partial \mathrm{a}^{(L)}}$$

$$\frac{\partial c}{\partial \mathrm{b}^{(L)}} = f(z^{(L)}) \cdot (1 - f(z^{(L)})) \cdot 2(a^{(L)} - y)$$

Now let's try an example of applying gradient descent, once again using the simple, 2 layer neural network from earlier.



We can also give each weight and bias a random numerical value to make it clearer as to if this process is working.

$w^{(0)} = 0.5 \qquad b^{(0)} = 1 \qquad w^{(1)} = 1 \qquad b^{(1)} = 0$

We should also have an input and the expected output (again this is random).

$x = 1 \qquad y = 0$

We also need a learning rate (although it will be 1 since we are only going to do 1 iteration, and I want to demonstrate a noticable change in the cost).

$\alpha = 1$

First, we should plug in the input to the neural network, to find out how accurate it currently is:

$z^{(0)} = w^{(0)} \cdot x + b^{(0)} = 0.5 \cdot 1 + 1 = 1.5$

$a^{(0)} = f\left(z^{(0)}\right) = \frac{1}{1 + e^{-(1.5)}} = 0.818$

$z^{(1)} = w^{(1)} \cdot a^{(0)} + b^{(1)} = 1 \cdot 0.818 + 0 = 0.818$

$\hat{y} = a^{(1)} = f\left(z^{(1)}\right) = \frac{1}{1 + e^{-(0.818)}} = 0.694$

$c = (\hat{y} - y)^2 = (0.694 - 0)^2 = 0.481$

As you can see, the cost is quite large, but let's see how that changes after an iteration of gradient descent.

First we find the derivative of the cost with respect to $w^{(1)}$ and $b^{(1)}$ (which we know from our equations for $w^{(L)}$ and $b^{(L)}$) and subtract this from $w^{(1)}$ and $b^{(1)}$ to find the new values for $w^{(1)}$ and $b^{(1)}$.

$$\frac{\partial c}{\partial w^{(1)}} = a^{(0)} \cdot f\left(z^{(1)}\right) \cdot \left(1 - f\left(z^{(1)}\right)\right) \cdot 2\left(a^{(1)} - y\right)$$
$$= 0.818 \cdot f(0.818) \cdot \left(1 - f(0.818)\right) \cdot 2(0.694 - 0) = 0.241$$

$$new\ w^{(1)} = old\ w^{(1)} - \alpha \cdot \frac{\partial c}{\partial w^{(1)}} = 1 - 1 \cdot 0.241 = 0.759$$

$$\frac{\partial c}{\partial b^{(1)}} = f\left(z^{(1)}\right) \cdot \left(1 - f\left(z^{(1)}\right)\right) \cdot 2\left(a^{(1)} - y\right) = f(0.818) \cdot \left(1 - f(0.818)\right) \cdot 2(0.694 - 0) = 0.295$$

$$new\ b^{(1)} = old\ b^{(1)} - \alpha \cdot \frac{\partial c}{\partial b^{(1)}} = 0 - 1 \cdot 0.295 = -0.295$$

Now we find $v^{(1)}$ to make calculating the derivatives for the next layer easier.

$$v^{(1)} = \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial c}{\partial a^{(1)}} = f\left(z^{(1)}\right) \cdot \left(1 - f\left(z^{(1)}\right)\right) \cdot 2\left(a^{(1)} - y\right)$$
$$= f(0.818) \cdot \left(1 - f(0.818)\right) \cdot 2(0.694 - 0) = 0.295$$

Next we repeat the process but for the weight and bias in layer 0.

$$\frac{\partial c}{\partial w^{(0)}} = x \cdot f\left(z^{(0)}\right) \cdot \left(1 - f\left(z^{(0)}\right)\right) \cdot w^{(1)} \cdot v^{(1)} = 1 \cdot f(1.5) \cdot \left(1 - f(1.5)\right) \cdot 1 \cdot 0.295 = 0.0440$$

$$new\ w^{(0)} = old\ w^{(0)} - \alpha \cdot \frac{\partial c}{\partial \mathrm{w}^{(0)}} = 0.5 - 1 \cdot 0.0440 = 0.456$$

$$\frac{\partial c}{\partial \mathrm{b}^{(0)}} = f\left(z^{(0)}\right) \cdot \left(1 - f\left(z^{(0)}\right)\right) \cdot w^{(1)} \cdot v^{(1)} = f(1.5) \cdot \left(1 - f(1.5)\right) \cdot 1 \cdot 0.295 = 0.0440$$

$$new\ b^{(0)} = old\ b^{(0)} - \alpha \cdot \frac{\partial c}{\partial \mathrm{b}^{(0)}} = 1 - 1 \cdot 0.0440 = 0.956$$

We've now obtained the updated values for the weights and biases in the neural network:

$w^{(0)} = 0.456 \qquad b^{(0)} = 0.956 \qquad w^{(1)} = 0.759 \qquad b^{(1)} = -0.295$

$$z^{(0)} = w^{(0)} \cdot x + b^{(0)} = 0.456 \cdot 1 + 0.956 = 1.412$$

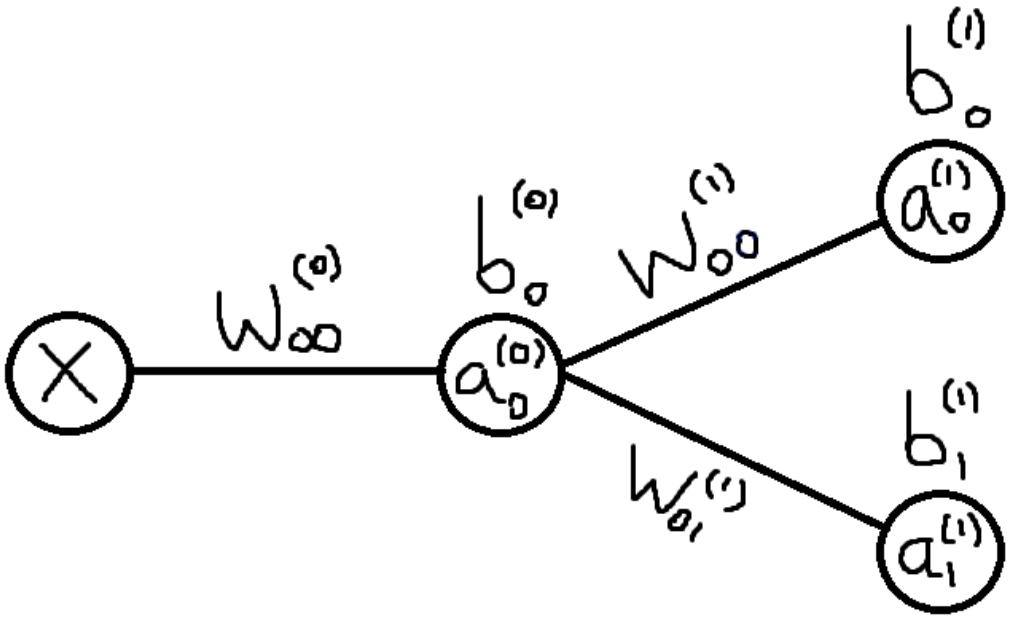
$$a^{(0)} = f\left(z^{(0)}\right) = \frac{1}{1 + e^{-(1.412)}} = 0.804$$

$$z^{(1)} = w^{(1)} \cdot a^{(0)} + b^{(1)} = 0.759 \cdot 0.804 - 0.295 = 0.315$$

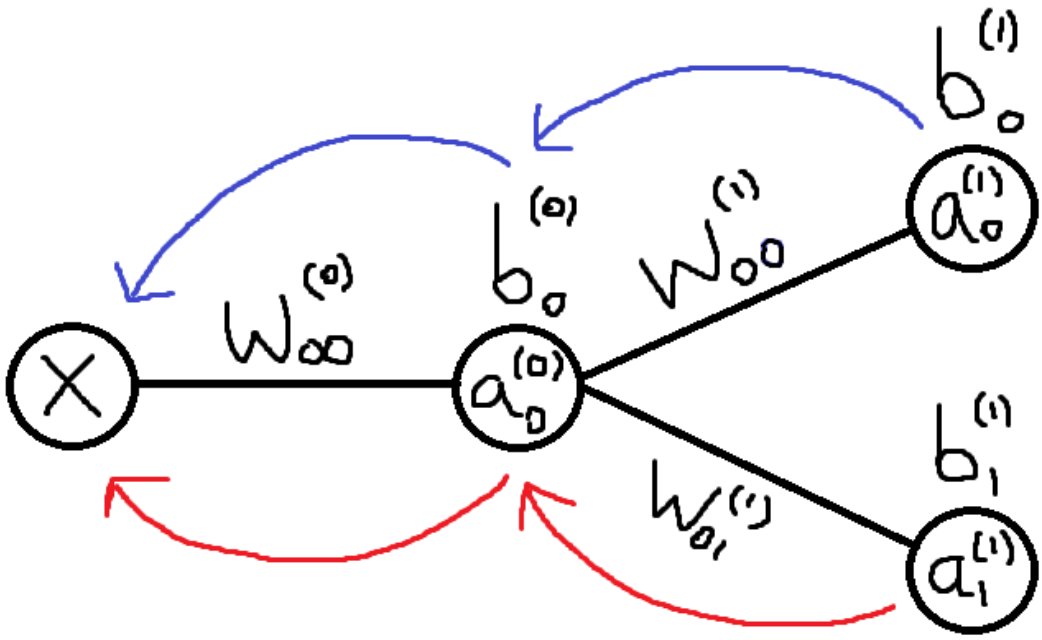
$$\hat{\mathrm{y}} = a^{(1)} = f\left(z^{(1)}\right) = \frac{1}{1 + e^{-(0.315)}} = 0.578$$

$$c = (\hat{\mathrm{y}} - \mathrm{y})^2 = (0.578 - 0)^2 = 0.334$$

And look at that! After just one iteration, our cost has noticeably decreased (from 0.481 to 0.334), and our neural network is just a bit more accurate. If we completed more iterations and gave it more data, the neural network would become more and more accurate as its cost decreases. The only thing left to do now is to make this actually useful by making the neural network larger.



Luckily, increasing the size of the neural network doesn't affect these calculations that much. The difference is shown here, as $w_{00}^{(0)}$ and $b_0^{(1)}$ affect not only $a_0^{(1)}$, but also $a_1^{(1)}$.



You can see here that there are two paths to get to $w_{00}^{(0)}$ from layer 1, and each path represents its own derivative with respect to $w_{00}^{(0)}$. This is easily accounted for by finding the sum of the different derivatives.

$$blue\ path = \frac{\partial z_0^{(0)}}{\partial w_{00}^{(0)}} \times \frac{\partial a_0^{(0)}}{\partial z_0^{(0)}} \times \frac{\partial z_0^{(1)}}{\partial a_0^{(0)}} \times \frac{\partial a_0^{(1)}}{\partial z_0^{(1)}} \times \frac{\partial c}{\partial a_0^{(1)}}$$

$$red\ path = \frac{\partial z_0^{(0)}}{\partial w_{00}^{(0)}} \times \frac{\partial a_0^{(0)}}{\partial z_0^{(0)}} \times \frac{\partial z_1^{(1)}}{\partial a_0^{(0)}} \times \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \times \frac{\partial c}{\partial a_1^{(1)}}$$

The first term, $\frac{\partial z_0^{(0)}}{\partial w_{00}^{(0)}}$, will always be the same across different paths, so that can be factored out when adding the two paths:

$$\frac{\partial c}{\partial w_{00}^{(0)}} = \frac{\partial z_0^{(0)}}{\partial w_{00}^{(0)}} \times \left(\frac{\partial a_0^{(0)}}{\partial z_0^{(0)}} \times \frac{\partial z_0^{(1)}}{\partial a_0^{(0)}} \times \frac{\partial a_0^{(1)}}{\partial z_0^{(1)}} \times \frac{\partial c}{\partial a_0^{(1)}} + \frac{\partial a_0^{(0)}}{\partial z_0^{(0)}} \times \frac{\partial z_1^{(1)}}{\partial a_0^{(0)}} \times \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \times \frac{\partial c}{\partial a_1^{(1)}} \right)$$

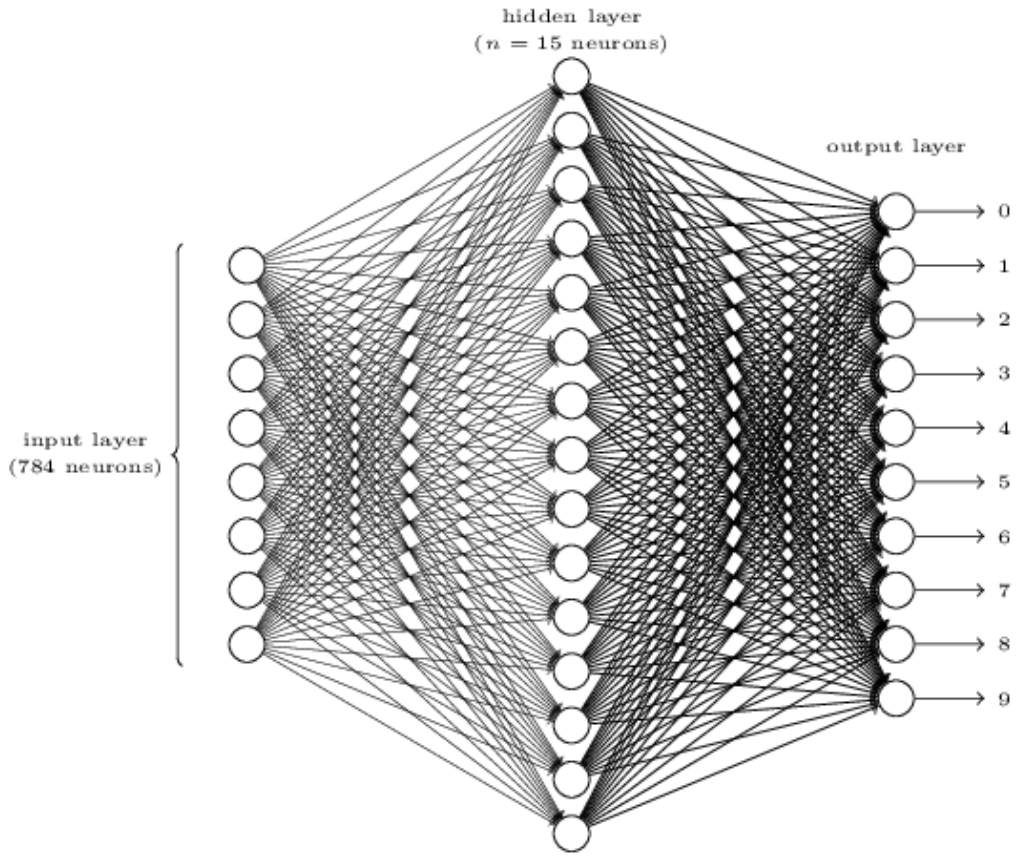
This equation seems long, but we know all these values and only simple multiplication and addition takes place, so a computer can perform it extremely quickly. As well as this, we can simplify by substituting $v_0^{(1)}$ and $v_1^{(1)}$ to give:

$$\frac{\partial c}{\partial w_{00}^{(0)}} = \frac{\partial z_0^{(0)}}{\partial w_{00}^{(0)}} \times \left(\frac{\partial a_0^{(0)}}{\partial z_0^{(0)}} \times \frac{\partial z_0^{(1)}}{\partial a_0^{(0)}} \times v_0^{(1)} + \frac{\partial a_0^{(0)}}{\partial z_0^{(0)}} \times \frac{\partial z_1^{(1)}}{\partial a_0^{(0)}} \times v_1^{(1)} \right)$$

Great! Now you know how backpropagation works and how it scales up to larger neural networks. Backpropagation is the driving force behind the learning process of a neural network, so understanding how this process means you finally understand how neural networks work.

Bringing it all together

We can now return to the problem suggested at the beginning of this essay: recognising handwritten digits. Let's say the input is a 28x28 pixel image of a handwritten digit, and the output should be the digit drawn in the image. For this, we would build a very large neural network with one input node for each pixel in the input image (28·28 = 784 input nodes), where the number in the node is represents the brightness value of the pixel (a value from 0 to 1). We would also need 10 output nodes, one node for each possible digit (this is because each node will display a value from 0 to 1 indicating how certain the neural network is that the image contains a certain digit, e.g. a value of 0.90 in the first output node means the neural network is 90% sure the image shows a handwritten 0). The final output of the neural network will be the digit with the highest probability of being correct. The network will also have one hidden layer with 15 nodes (although this decision is fairly arbitrary).



From this point, we simply train the neural network with thousands of handwritten images, fine tuning each weight and bias in order to create the perfect function which takes in 784 brightness values, and outputs the correct digit.

And hopefully at this point you can understand how a neural network is really just a function approximator. In some cases (such as this one), it approximates huge, incomprehensible functions with hundreds of variables, but it still is just a function.

Machine learning is a topic I feel is misunderstood by many people, who believe it to be wizardry and beyond comprehension. I'm hoping that over the course of this essay, not only have I been able to dispel some of the magic surrounding machine learning and neural networks, but also help you to appreciate how even relatively simple mathematics can be extraordinarily useful for revolutionary technology.

References

Handwritten 7 image: https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

Neural network for recognising handwritten digits image: http://neuralnetworksanddeeplearning.com/chap1.html

https://www.youtube.com/watch?v=tIeHLnjs5U8

https://www.youtube.com/watch?v=hfMk-kjRv4c

https://www.youtube.com/watch?v=0QczhVg5HaI&t=316s