How Do We Actually Update Weights?

A neural network's error is a function of the internal link weight. Improving a neural network means reducing this error - by changing the weights. But how do we change those weights? Let's find out!

> We have not yet attacked the very central question of updating the link weights in a neural network. We've been working to get to this point, and we're almost there. We have just one more key idea to understand before we can unlock this secret. So far, we've got the errors propagated back to each layer of the network. Why did we do this? Because the error is used to guide how we adjust the link weights to improve the overall answer given by the neural network. This is basically what we were doing way back with the linear classifier at the start of this guide. But these nodes aren't simple linear classifiers. These slightly more sophisticated nodes sum the weighted signals into the node and apply the sigmoid threshold function. So how do we actually update the weights for links that connect these more sophisticated nodes? Why can't we use some fancy algebra to directly work out what the weights should be?
>
> We can't do fancy algebra to work out the weights directly because the maths is too hard. There are just too many combinations of weights and too many functions of functions of functions ... being combined when we feed forward the signal through the network. Think about even a small neural network with three layers and three neurons in each layer, as we had above. How would you tweak a weight for a link between the first input node and the second hidden node so that the third output node increased its output by, say, 0.5? Even if we did get lucky, the effect could be ruined by tweaking another weight to improve a different output node. You can see this isn't trivial at all.
>
> To see how untrivial, just look at the following horrible expression showing an output node's output as a function of the inputs and the link weights for a simple three-layer neural network with three nodes in each layer. The input at node i is x_i, the weights for links connecting input node i to hidden node j is $w_{i,j}$, similarly the output of hidden node j is x_j, and the weights for links

connecting hidden node j to output node k is $w_{j,k}$. That funny symbol $\sum a^b$ means to sum the subsequent expression for all values between a and b.

$$o_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}})}}$$

Yikes! Let's not untangle that. Instead of trying to be too clever, we could just simply try random combinations of weights until we find a good one? That's not always such a crazy idea when we're stuck with a hard problem. The approach is called a *Brute Force* method. Some people use brute force methods to try to crack passwords, and it can work if your password is an English word and not too long, because there aren't too many for a fast home computer to work through. Now, imagine that each weight could have 1000 possibilities between -1 and $+1$, like 0.501, -0.203 and 0.999 for example. Then for a three-layer neural network with three nodes in each layer, there are 18 weights, so we have $18,000$ possibilities to test. If we have a more typical neural network with 500 nodes in each layer, we have 500 million weight possibilities to test. If each set of combinations took 1 second to calculate, this would take us 16 years to update the weights after just one training example! A thousand training examples, and we'd be at $16,000$ years!