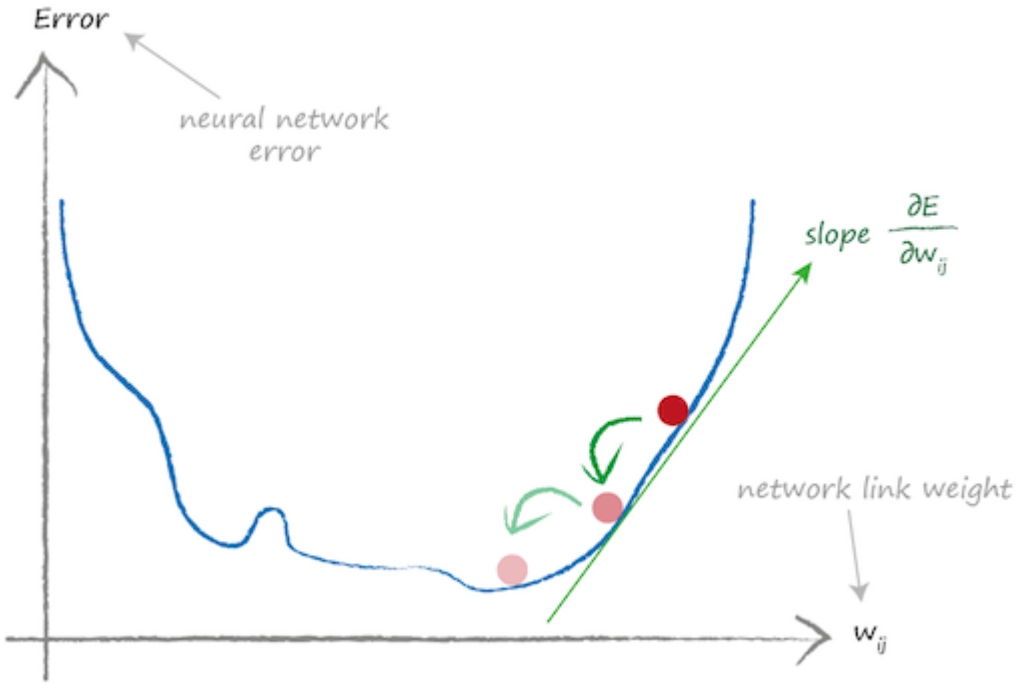
Using Gradient Descent to Update Weights

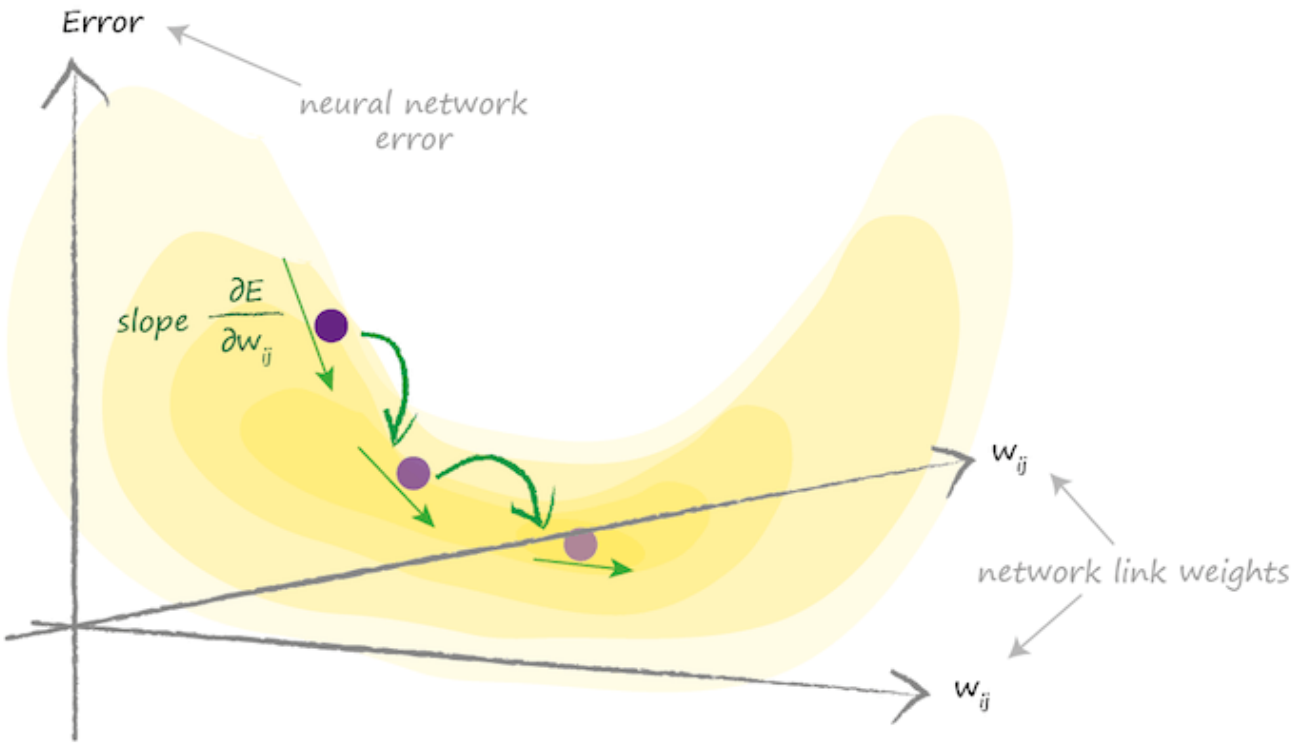
Now that we have designed the function, we will pass it to gradient descent algorithm to update the weights in our network.

To do gradient descent, we now need to work out the slope of the error function with respect to the weights. This requires *Calculus*. You may already be familiar with calculus, but if you're not, or just need a reminder, the *Appendix* contains a gentle introduction. Calculus is simply a mathematically precise way of working out how something changes when something else does. For example, how the length of spring changes as the force used to stretch it changes. Here we're interested in how the error function depends on the link weights inside a neural network. Another way of asking this is - "how sensitive is the error is to changes in the link weights?". Let's start with a picture because that always helps keep us grounded in what we are trying to achieve.



The graph is just like the one we saw before to emphasize that we're not doing anything different. This time the function we're trying to minimize is the neural network's error. The parameter we're trying to refine is a network link weight. In this simple example, we've only shown one weight, but we know

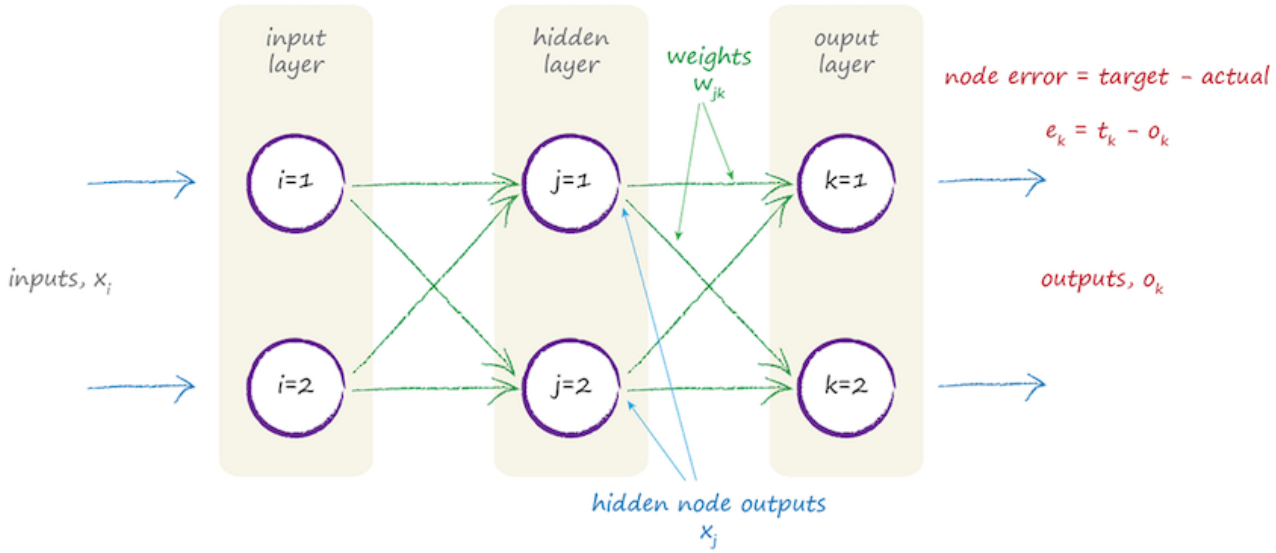
neural networks will have many more. The next diagram shows two link weights, and this time the error function is a 3-dimensional surface which varies as the two link weights vary. You can see we're trying to minimize the error which is now more like a mountainous landscape with a valley.



It's harder to visualize that error surface as a function of many more parameters, but the idea to use gradient descent to find the minimum is still the same. Let's write out mathematically what we want.

$$\frac{\partial E}{\partial w_{jk}}$$

That is, how does the error **E** change as the weight w_{jk} changes. That's the slope of the error function that we want to descend towards the minimum. Before we unpack that expression, let's focus for the moment only on the link weights between the hidden and the final output layers. The following diagram shows this area of interest highlighted. We'll come back to the link weights between the input and hidden layers later.



We will keep referring back to this diagram to make sure we don't forget what each symbol really means as we do the calculus. Don't be put off by it, the steps aren't difficult and will be explained, and all of the concepts needed have already been covered earlier. First, let's expand that error function, which is the sum of the differences between the target and actual values squared, and where that sum is over all the n output nodes.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_n (t_n - o_n)^2$$

All we have done here is write out what the error function **E** actually is. We can simplify this straight away by noticing that the output at a node n, which is o_n, only depends on the links that connect to it. That means for a node k, the output o_k only depends on weights w_{jk}, because those weights are for links into node k.

Another way of looking at this is that the output of a node k does not depend on weights w_{jb}, where b does not equal k because there is no link connecting them. The weight w_{jb} is for a link connecting to output node b not k. This means we can remove all the o_n from that sum except the one that the weight w_{jk} links to, that is o_k. This removes that pesky sum totally! A nice trick worth keeping in your back pocket. We will continue this discussion in the next lesson, stay tuned!