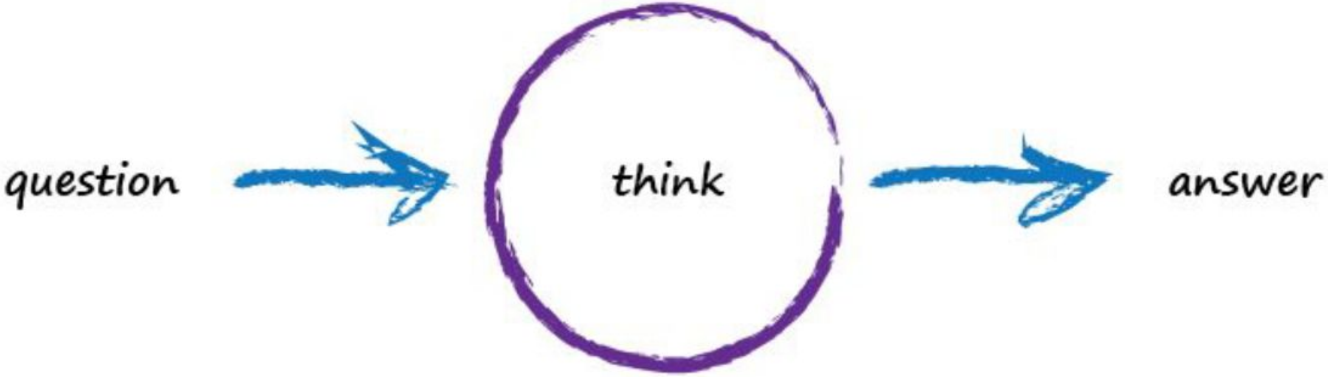
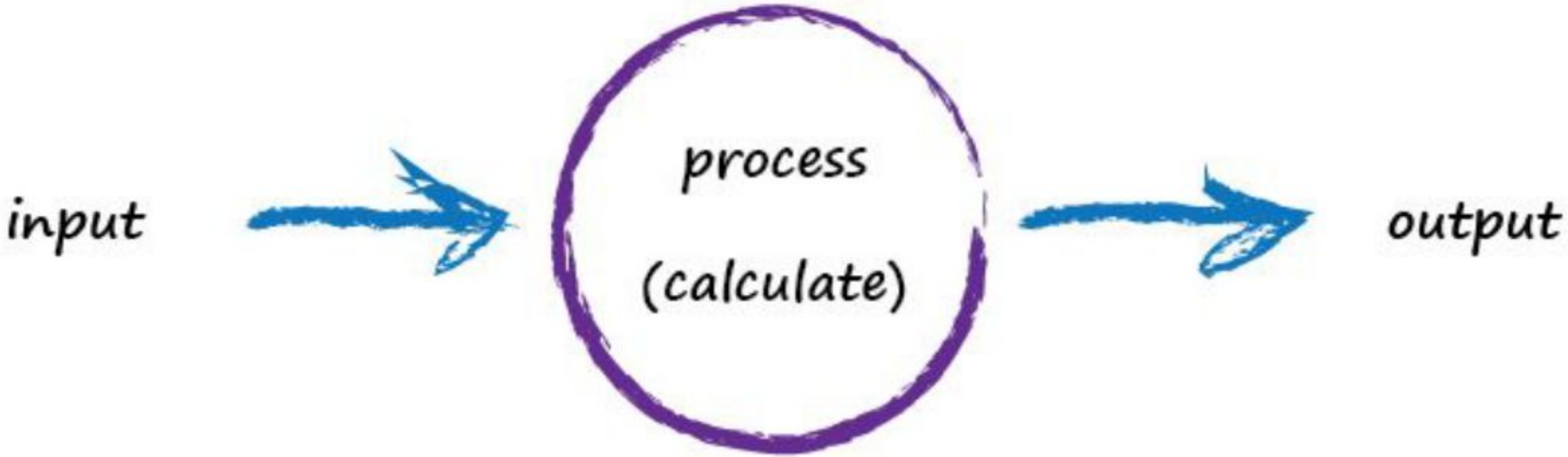
Fundamentals of Neural Networks

A Simple Predicting Machine

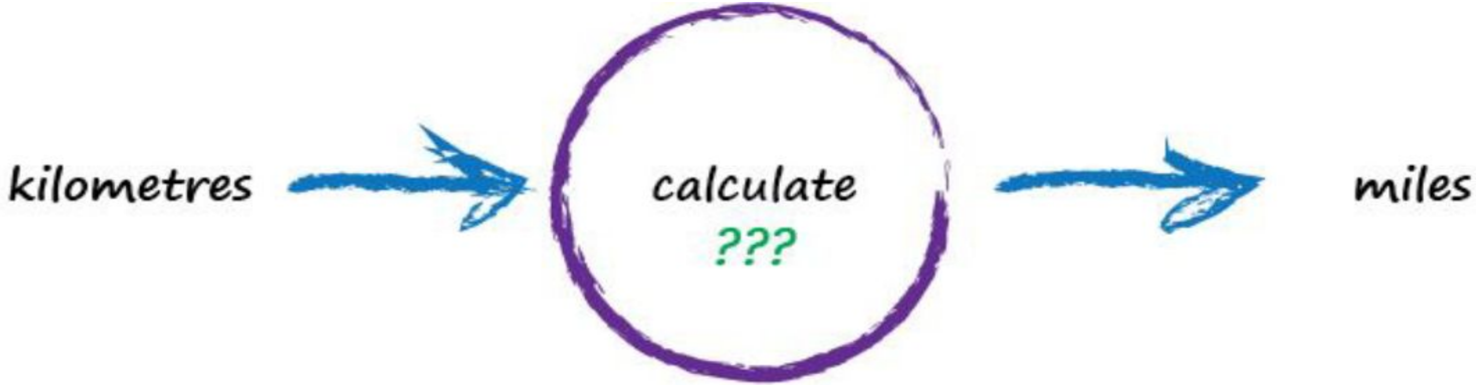
Imagine a basic machine that takes a question, does some “thinking” and pushes out an answer. Here’s what this looks like



Computers don't really think, they're just glorified calculators remember, so let's use more appropriate words to describe what's going on:



Imagine a machine that converts kilometres to miles, like the following:

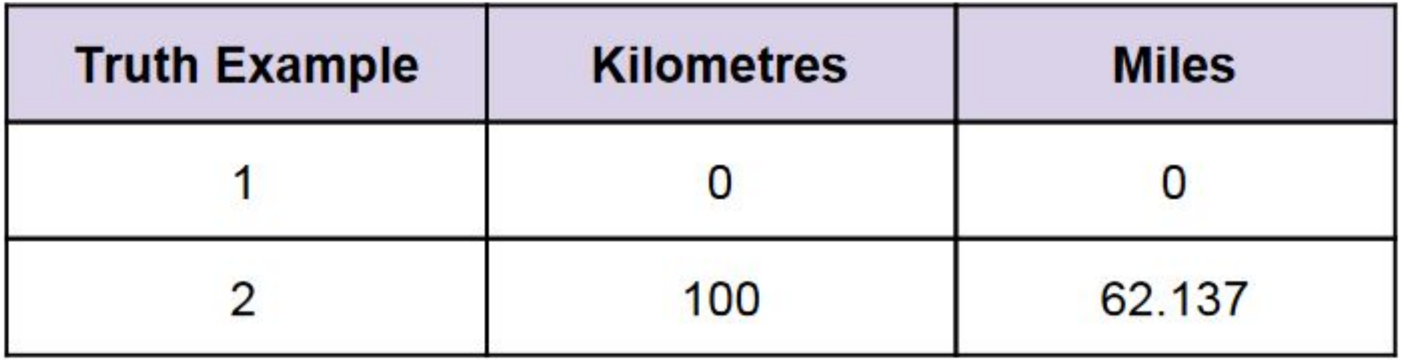


- All we know is the the relationship between the two is **linear**. That means if we double the number in miles, the same distance in kilometres is also doubled. That makes intuitive sense.

- This linear relationship between kilometres and miles gives us a clue about that mysterious calculation it needs to be of the form “miles = kilometres x c”, where c is a constant.

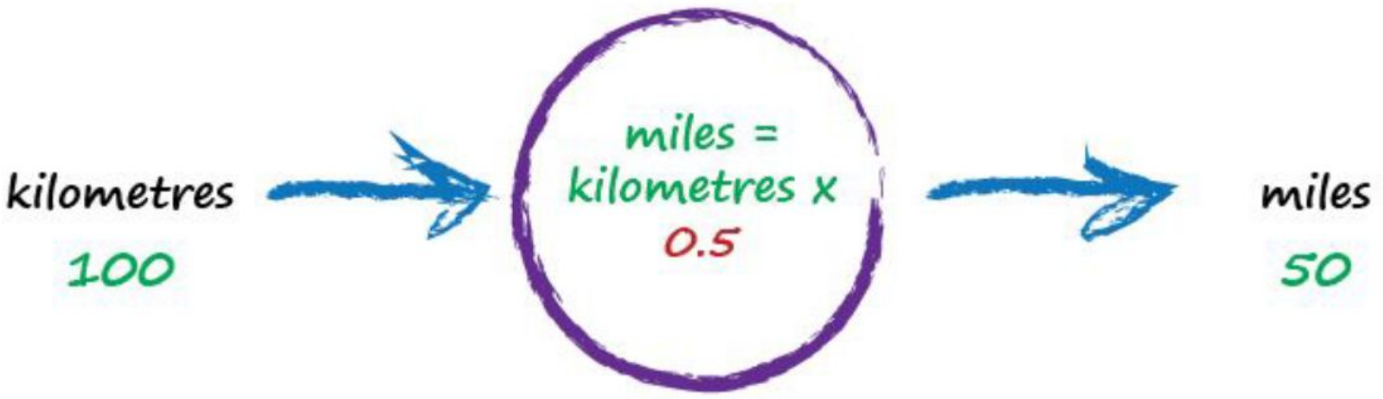
- We don’t know what this constant c is yet.

Example



Truth Example	Kilometres	Miles
1	0	0
2	100	62.137

What should we do to work out that missing constant c? Let’s just pluck a value at random and give it a go! Let’s try c = 0.5 and see what happens.



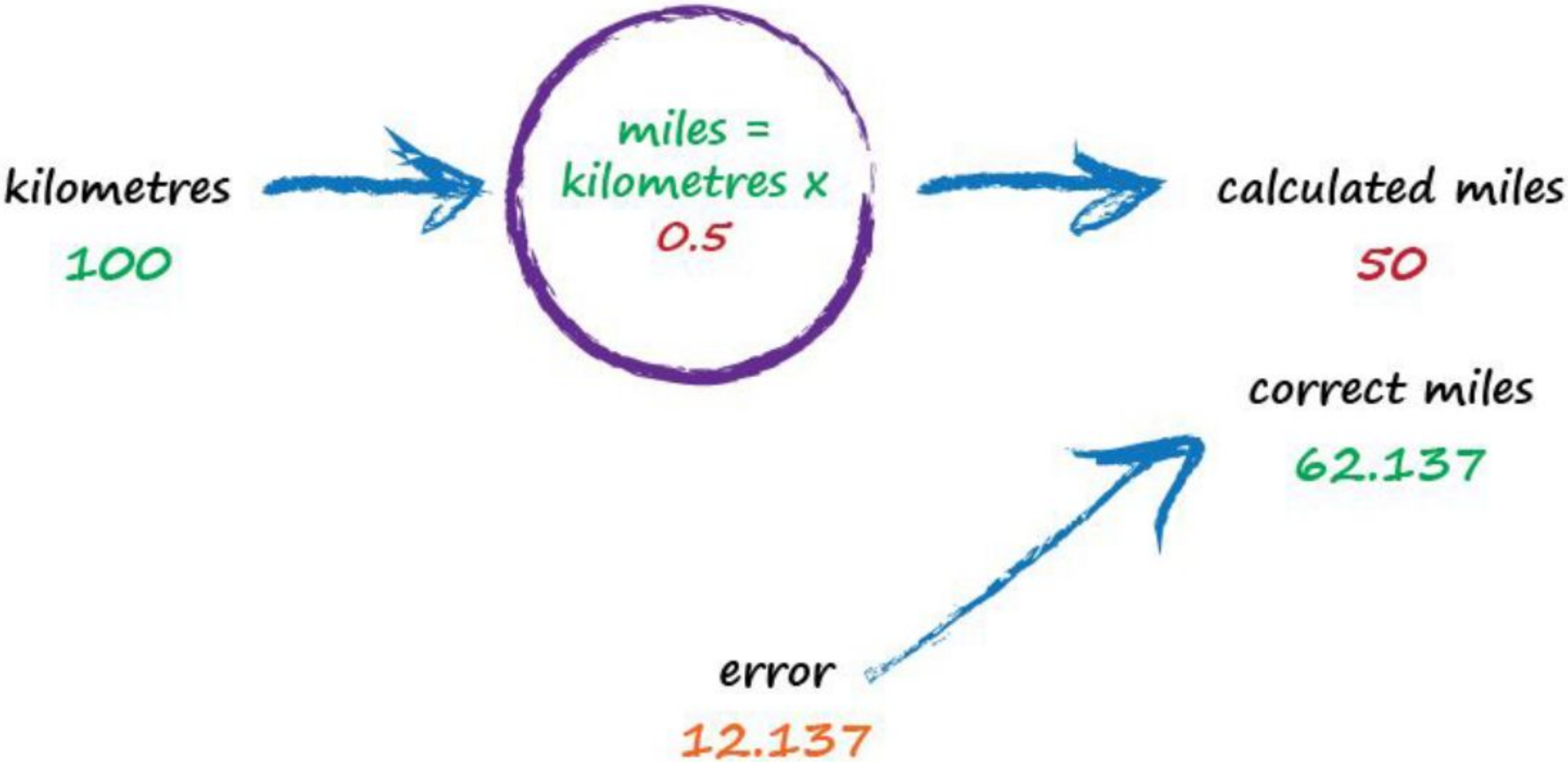
That’s not bad at all given we chose c = 0.5 at random! But we know it’s not exactly right because our truth example number 2 tells us the answer should be 62.137.

We’re wrong by 12.137. That’s the error, the difference between our calculated answer and the actual truth from our list of examples. That is,

```
error = truth - calculated

      = 62.137 - 50

      = 12.137
```



kilometres

100

miles =
kilometres x
0.5

calculated miles

50

correct miles

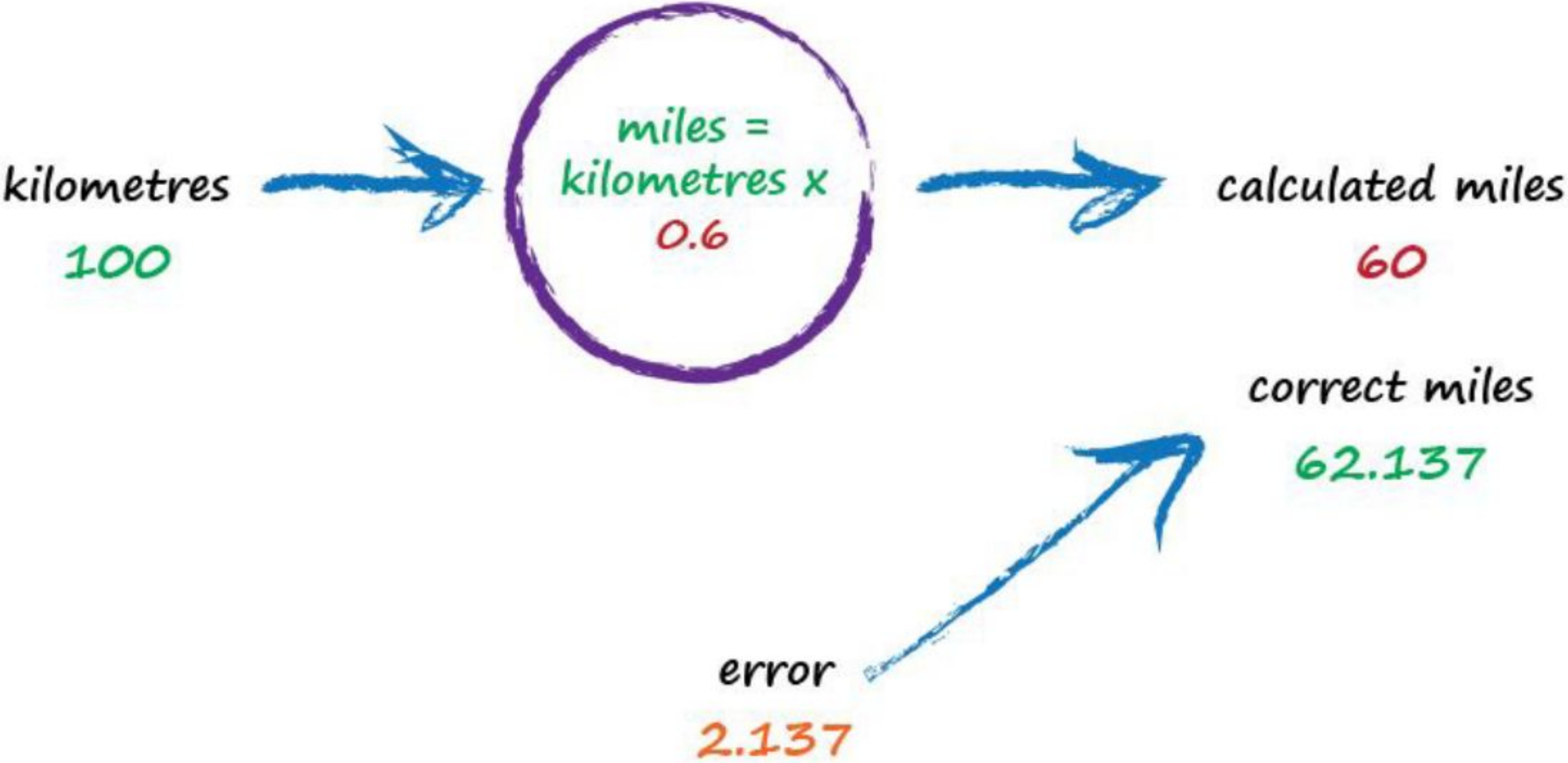
62.137

error

12.137

So what next?

- We know we’re wrong, and by how much. Instead of being a reason to despair,
- We use this error to guide a second, better, guess at c.
- Look at that error again. We were short by 12.137.
- We know that increasing c will increase the output.
- Let’s nudge c up from 0.5 to 0.6 and see what happens



kilometres

100

miles = kilometres x 0.6

calculated miles

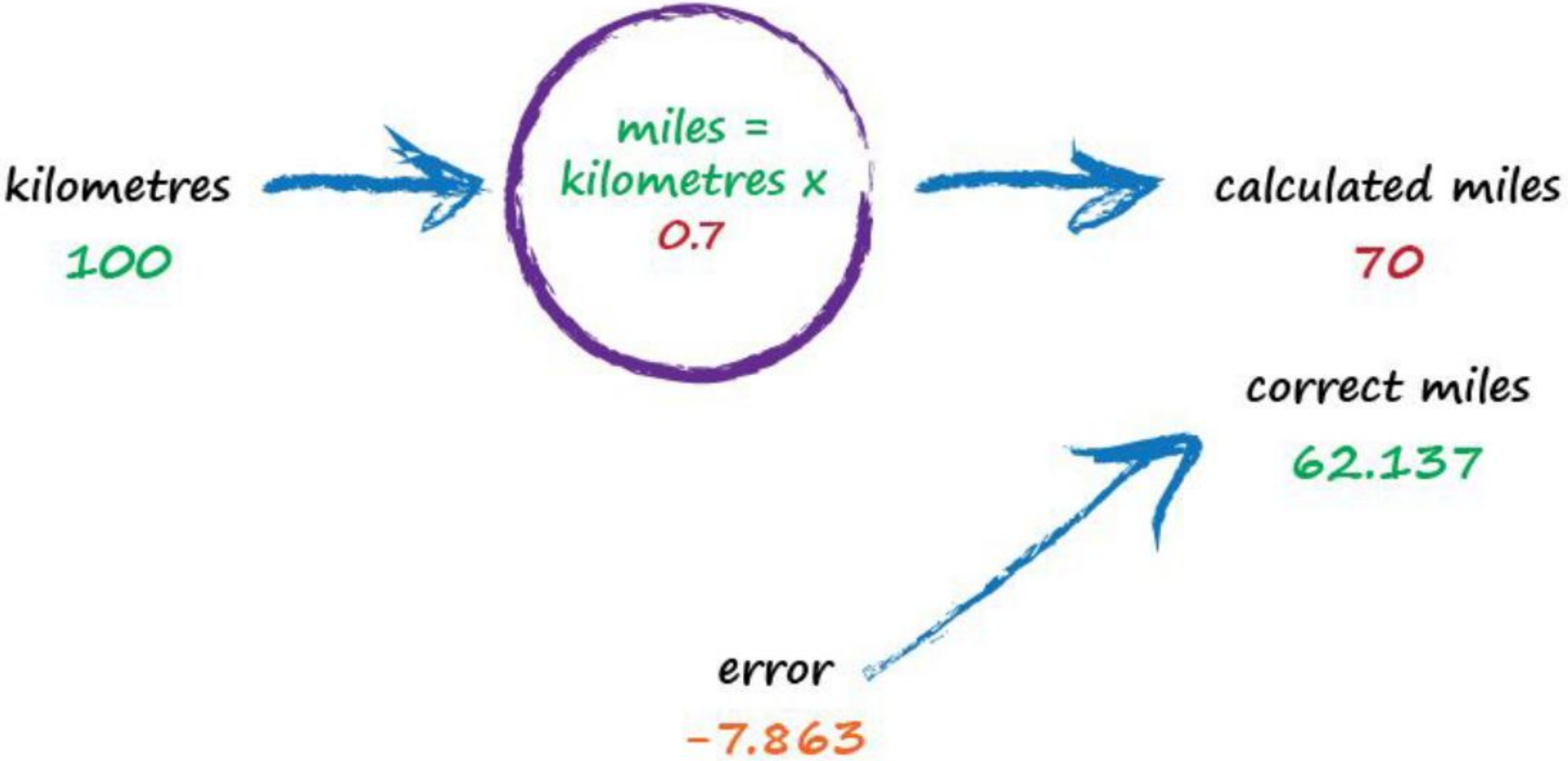
60

correct miles

62.137

error

2.137



kilometres

100

miles = kilometres x 0.7

calculated miles

70

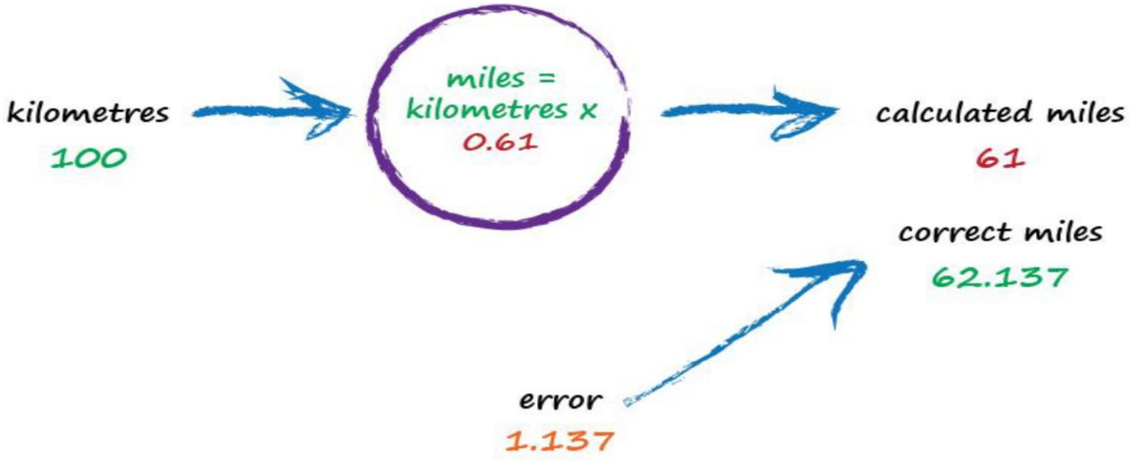
correct miles

62.137

error

-7.863

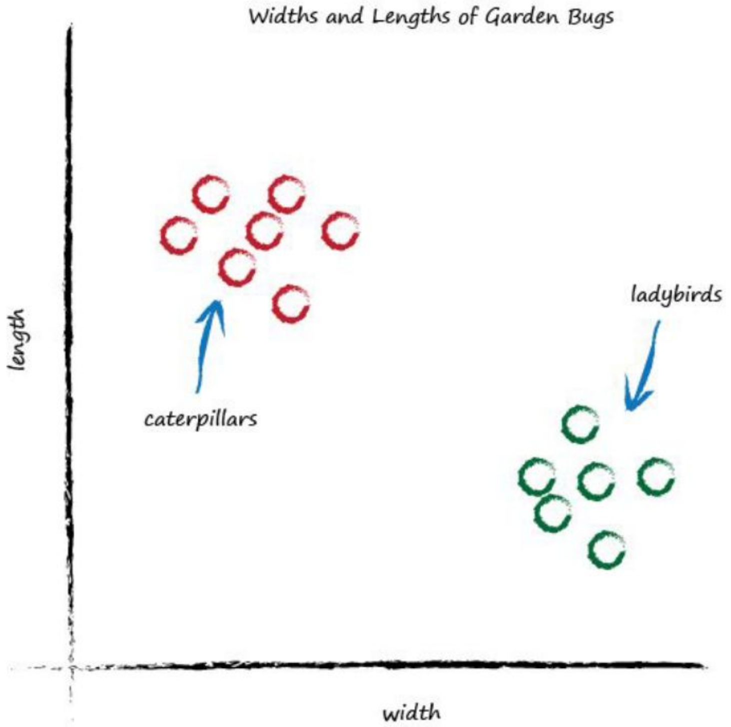
- We’ve gone too far and overshot the known correct answer.
- Our previous error was 2.137 but now it’s 7.863.
- The minus sign simply says we overshot rather than undershot, remember the error is (correct value calculated value).



- So that l**ast effort taught** us that we should moderate how much we nudge the value of c.
- Without getting too distracted by exact ways of working out c, and to remain focussed on this idea of successively refining it, we could suggest that the **correction is a fraction of the error**.
- That’s intuitively right a big error means a bigger correction is needed, and a tiny error means we need the teeniest of nudges to c.

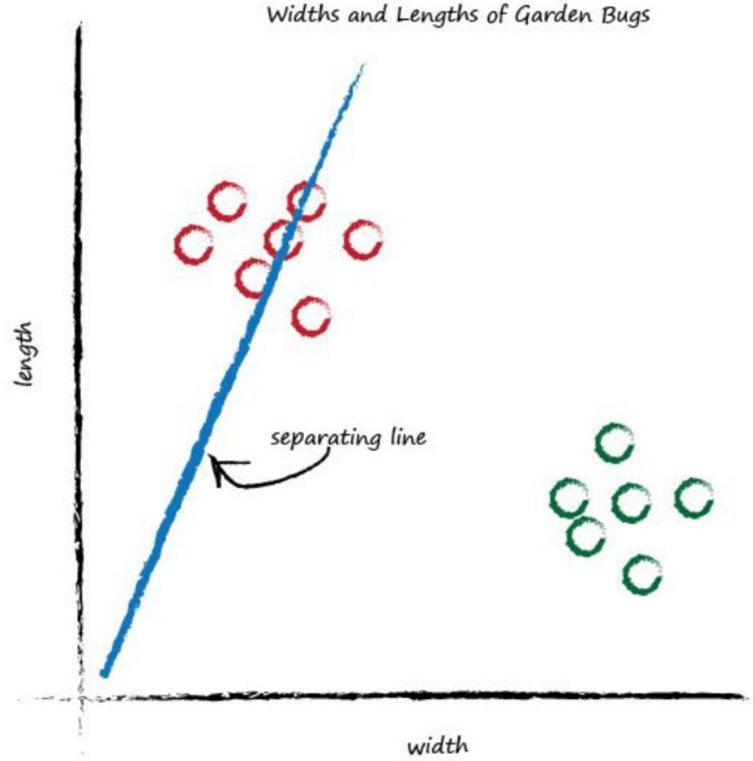
What we’ve just done, believe it or not, is walked through the very core process of learning in a neural network we’ve trained the machine to get better and better at giving the right answer.

Classifying is Not Very Different from Predicting

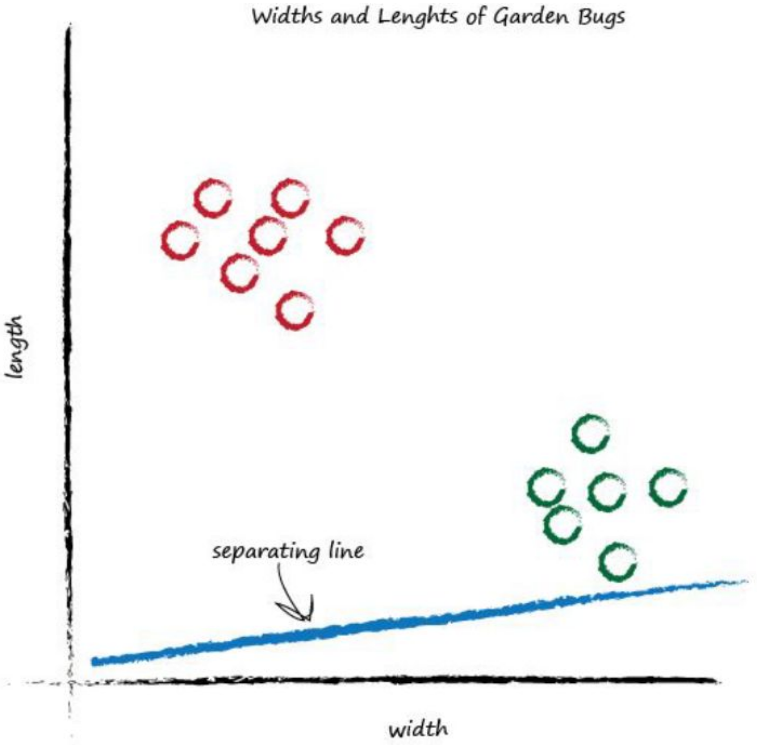


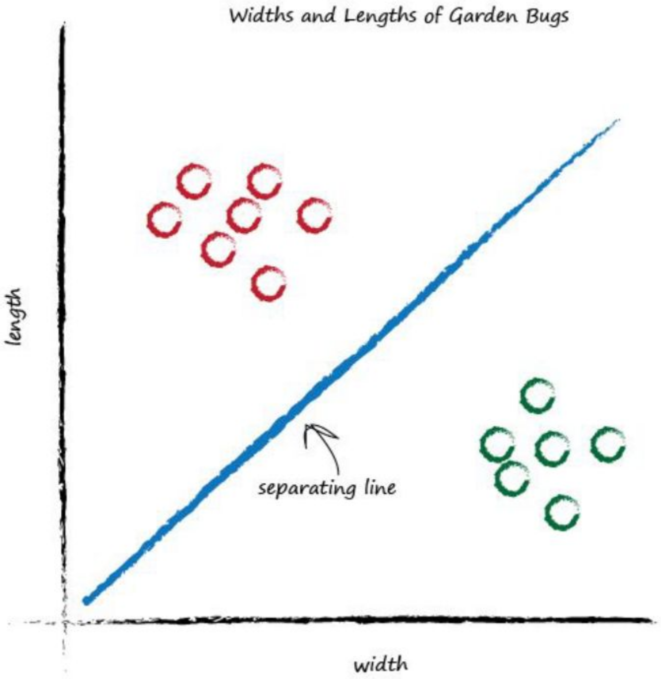
- We called the above simple machine a predictor, because it takes an input and makes a prediction of what the output should be.
- We refined that prediction by adjusting an internal parameter, informed by the error we saw when comparing with a known true example.
- Now look at the previous graph showing the measured widths and lengths of garden bugs.

What happens if we place a straight line over that plot?



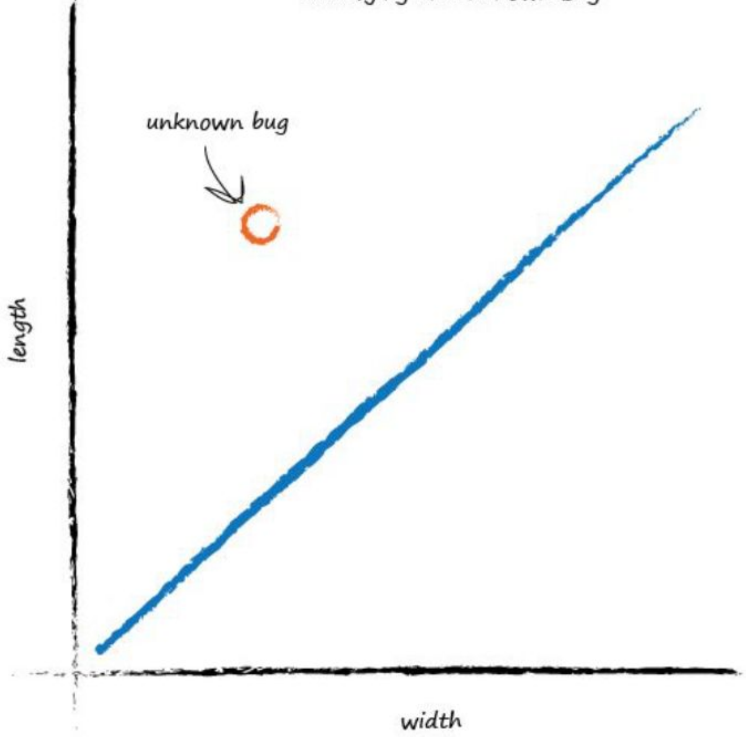
Let’s try a different line, by adjusting the slope again, and see what happens.





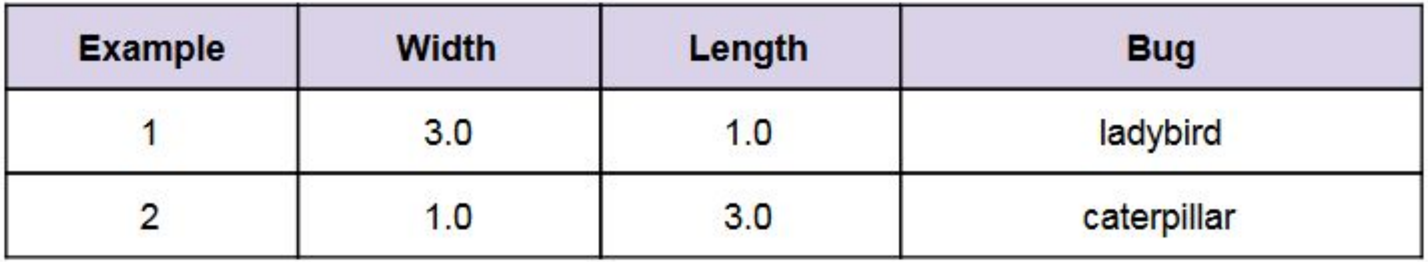
- That’s much better! This line neatly separates caterpillars from ladybirds. We can now use this line as a classifier of bugs.
- We are assuming that there are no other kinds of bugs that we haven’t seen but that’s ok for now, we’re simply trying to illustrate the idea of a simple classifier.
- Imagine next time our computer used a robot arm to pick up a new bug and measured its width and height, it could then use the above line to classify it correctly as a caterpillar or a ladybird.

Classifying an Unknown Bug

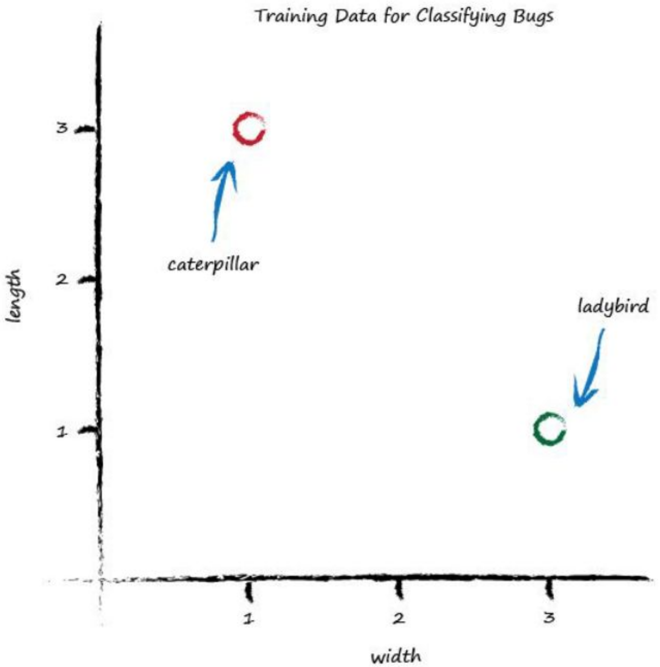


Training A Simple Classifier

We want to train our linear classifier to correctly classify bugs as ladybirds or caterpillars. We saw above this is simply about refining the slope of the dividing line that separates the two groups of points on a plot of big width and height.



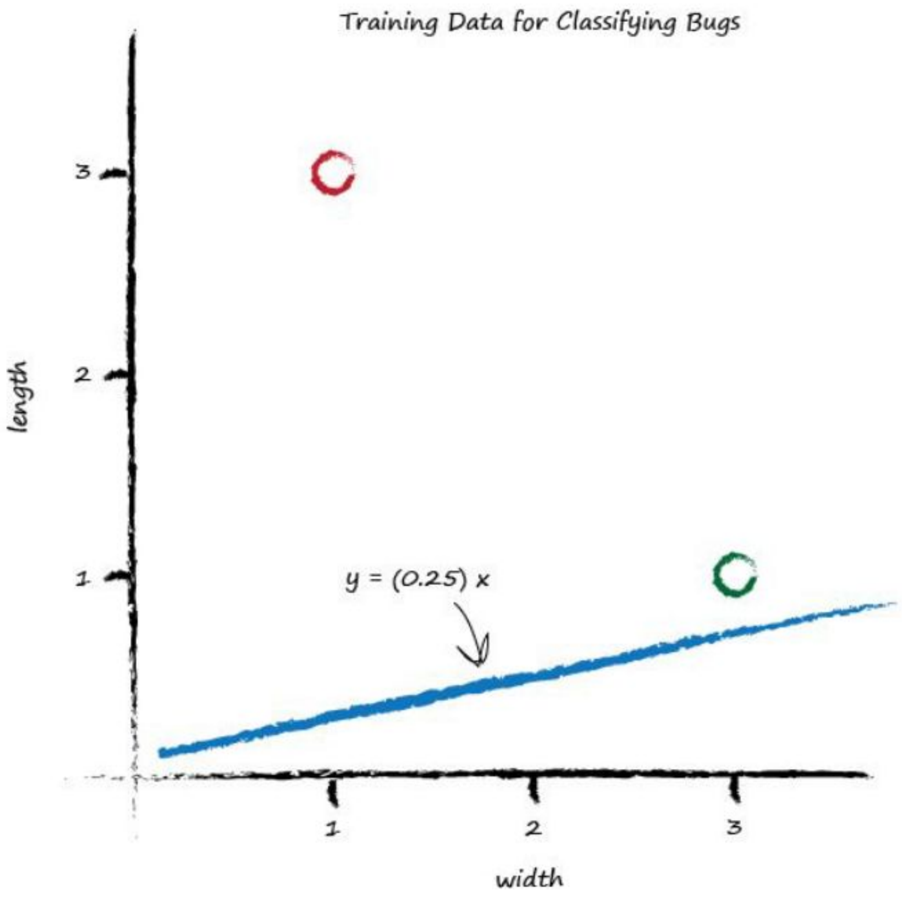
Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar



Let’s start with a random dividing line, just to get started somewhere. Looking back at our miles to kilometre predictor, we had a linear function whose parameter we adjusted. We can do the same here, because the dividing line is a straight line: **`y = Ax`**

We know that the parameter A controls the slope of the line. The larger A is the larger the slope.

Let’s go for A is 0.25 to get started. The dividing line is $y = 0.25x$. Let’s plot this line on the same plot of training data to see what it looks like:


Training Data for Classifying Bugs
length
3
2
1
y = (0.25) x
1
2
3
width

Let’s look at the first training example: the width is 3.0 and length is 1.0 for a ladybird. If we tested the y = Ax function with this example where x is 3.0, we’d get

```
y = (0.25) * (3.0) = 0.75
```

The function, with the parameter A set to the initial randomly chosen value of 0.25, is suggesting that for a bug of width 3.0, the length should be 0.75. We know that’s too small because the training data example tells us it must be a length of 1.0.

So we have a difference, an error. Just as before, with the miles to kilometres predictor, we can use this error to inform how we adjust the parameter A.

- But before we do, let’s think about what y should be again.
- If y was 1.0 then the line goes right through the point where the ladybird sits at (x,y) = (3.0, 1.0). It’s a subtle point but we don’t actually want that.
- We want the l**ine to go above that point**. Why? Because we want all the ladybird points to be below the line, not on it.
- **The line needs to be a dividing line between ladybirds and caterpillars, not a predictor of a bug’s length given its width.**
- So let’s try to aim for y = 1.1 when x = 3.0.

So the desired target is 1.1, and the error E is

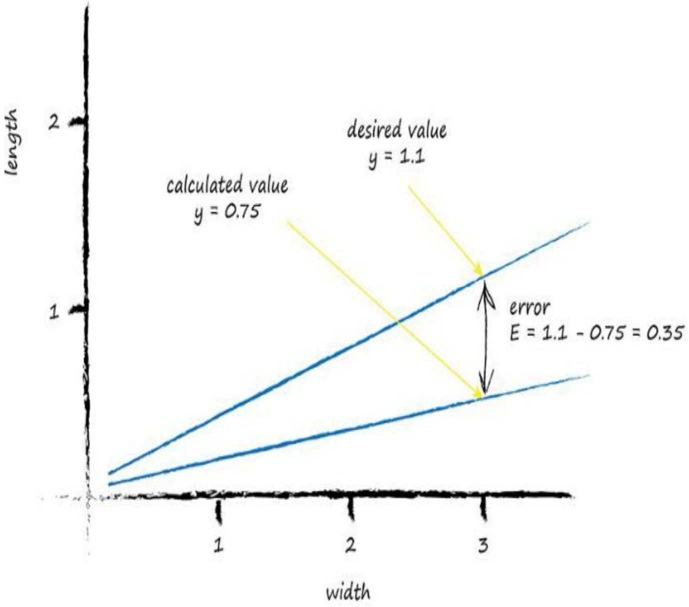
`error = (desired target - actual output)`

Which is,

`E = 1.1 - 0.75 = 0.35`

Let’s pause and have a remind ourselves what the

and the calculated value mean visually.



Now, what do we do with this E to guide us to a better refined parameter A? That’s the important question.

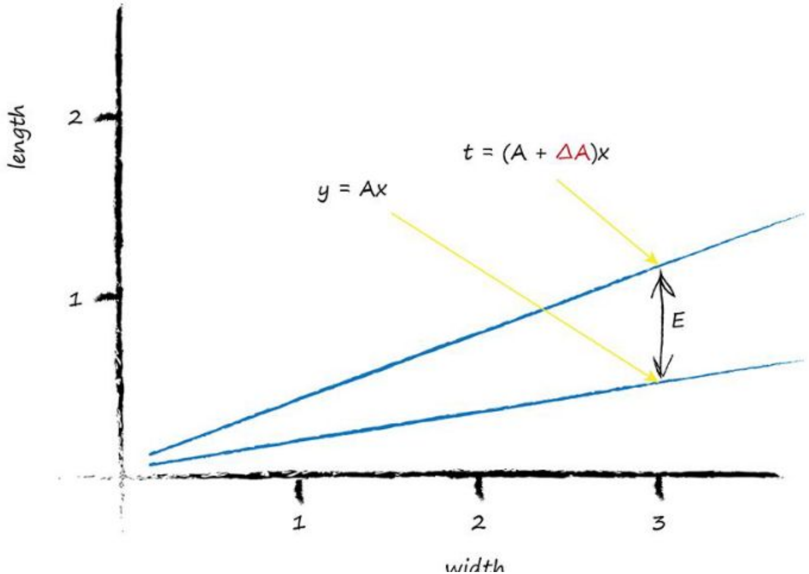
- Let’s take a step back from this task and think again. We want to use the error in y, which we call E, to inform the required change in parameter A.
- To do this we need to know how the two are related. How is A related to E?
- Let’s start with the linear function for the classifier:

$$\mathbf{y = Ax}$$

- We know that for initial guesses of A this gives the wrong answer for y, which should be the value given by the training data. Let’s call the correct desired value, t for target value.
- To get that value t, we need to adjust A by a small amount.

Mathematicians use the d**elta symbol Δ to mean “a small change in**”. Let’s write that out:

`t = (A + ΔA)x`



You can see the new slope (A + ΔA).

Remember the error E was the difference between the desired correct value and the one we calculate based on our current guess for A.

Let’s write that out to make it clear:

$$t - y = (A + \Delta A)x - Ax$$

Expanding out the terms and simplifying:

$$E = t - y = Ax + (\Delta A)x - Ax$$

$$E = (\Delta A)x$$

This simple relationship makes our job much easier.

How much to adjust A by to improve the slope of the line so it is a better classifier?

Simply re-arrange that last equation

$$\Delta A = E / x$$

The error was 0.35 and the x was 3.0.

$\Delta A = E / x$ as $0.35 / 3.0 = 0.1167$.

That means we need to change the current A = 0.25 by 0.1167.

The new improved value for A is

$(A + \Delta A)$ which is $0.25 + 0.1167 = 0.3667$.

The calculated value of y with this new A is 1.1 as you’d expect it’s the desired target value.

Let’s see what happens when we put x = 1.0 into the linear function which is now using the updated A = 0.3667.

We get y = 0.3667 * 1.0 = 0.3667. That’s not very close to the training example with y = 3.0 at all.

Using the same reasoning as before we can set the desired target value at 2.9. **This way the training example of a caterpillar is just above the line, not on it.** The error E is (2.9 - 0.3667) = **2.5333.**

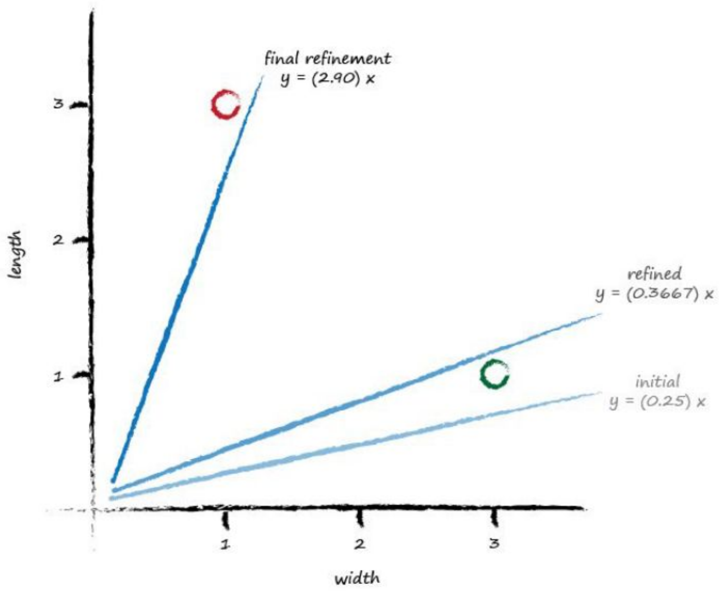
Bigger error

Let’s update the A again, just like we did before.

The ΔA is E / x which is 2.5333 / 1.0 = 2.5333.

That means the even newer A is 0.3667 + 2.5333 = **2.9**

desired value



What’s wrong with that?

- Well, if we keep doing this, updating for each training data example, all we get is that the final update simply matches the last training example closely.
- We might as well have not bothered with all previous training examples.
- In effect we are throwing away any learning that previous training examples might gives us and just learning from the last one.

How do we fix this?

Easy! And this is an important idea in machine learning.

We moderate the updates.

Instead of jumping enthusiastically to each new A, we take a fraction of the change ΔA, not all of it.

This moderation, has another very powerful and useful side effect. When the training data itself can’t be trusted to be perfectly true, and contains errors or noise, the moderation can dampen the impact of those errors or noise. It smooths them out.

Ok let’s rerun that again, but this time we’ll add a moderation into the update formula:

$$\Delta A = L \ (E \ / \ x \ )$$

The moderating factor is often called a learning rate, and we’ve called it L.

Let’s pick L = 0.5 as a reasonable fraction just to get started.

It simply means we only update half as much as would have done without moderation.

- Running through that all again, we have an initial A = 0.25.
- The first training example gives us y = 0.25 * 3.0 = 0.75.
- A desired value of 1.1 gives us an error of 0.35.
- The ΔA = L (E / x) = 0.5 * 0.35 / 3.0 = 0.0583.
- The updated A is 0.25 + 0.0583 = 0.3083.
- Trying out this new A on the training example at x = 3.0 gives y = 0.3083 * 3.0 = 0.9250.
- The line now falls on the wrong side of the training example because it is below 1.1.

Not Bad

Let’s press on to the second training data example at x = 1.0.

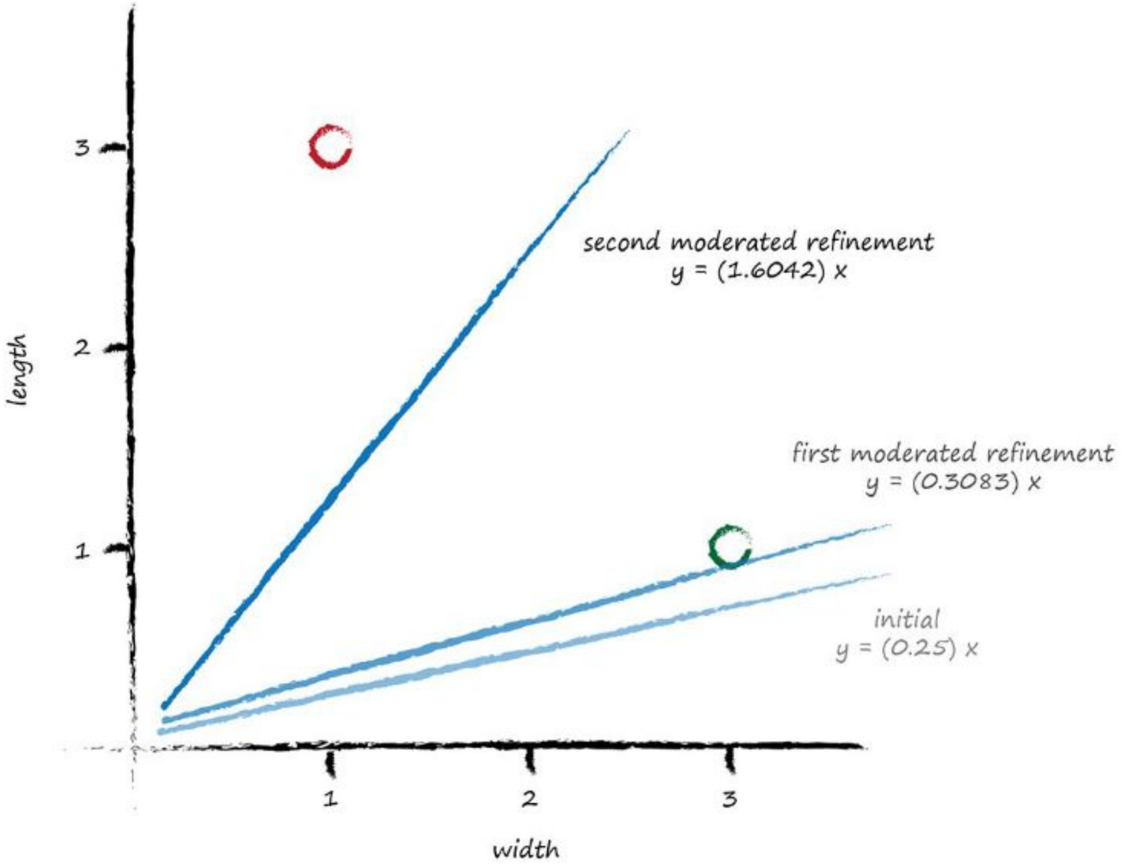
Using A = 0.3083 we have y = 0.3083 * 1.0 = 0.3083.

The desired value was 2.9 so the error is (2.9 - 0.3083) = 2.5917.

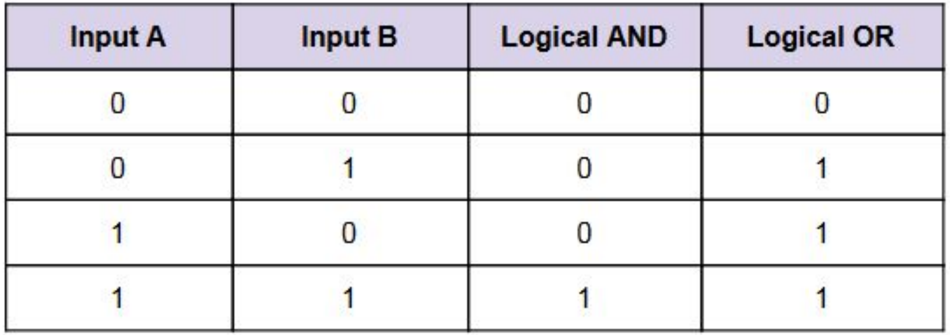
The ΔA = L (E / x) = 0.5 * 2.5917 / 1.0 = 1.2958.

The even newer A is now 0.3083 + 1.2958 = 1.6042

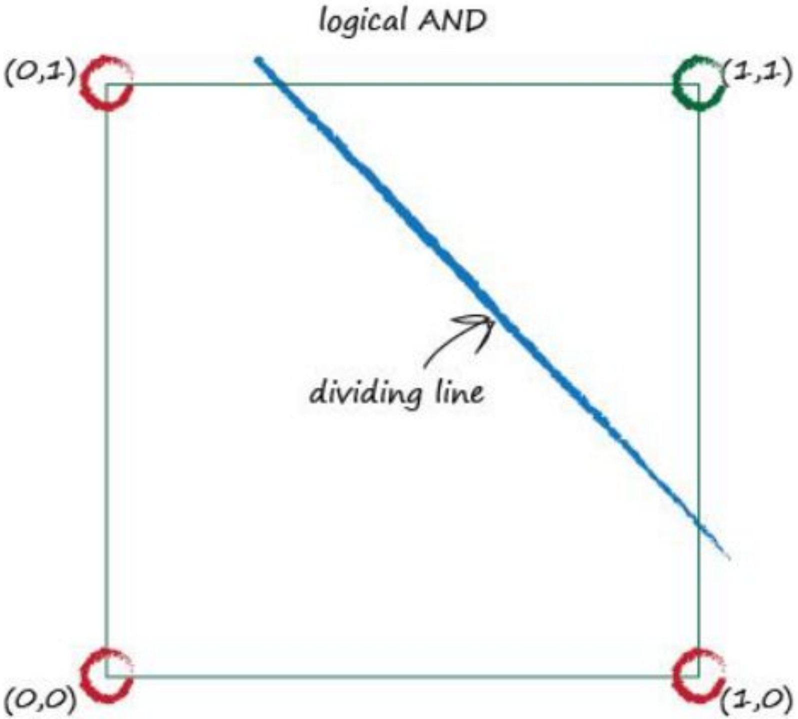
We’ve achieved an automated method of learning to classify from examples that is remarkably effective given the simplicity of the approach

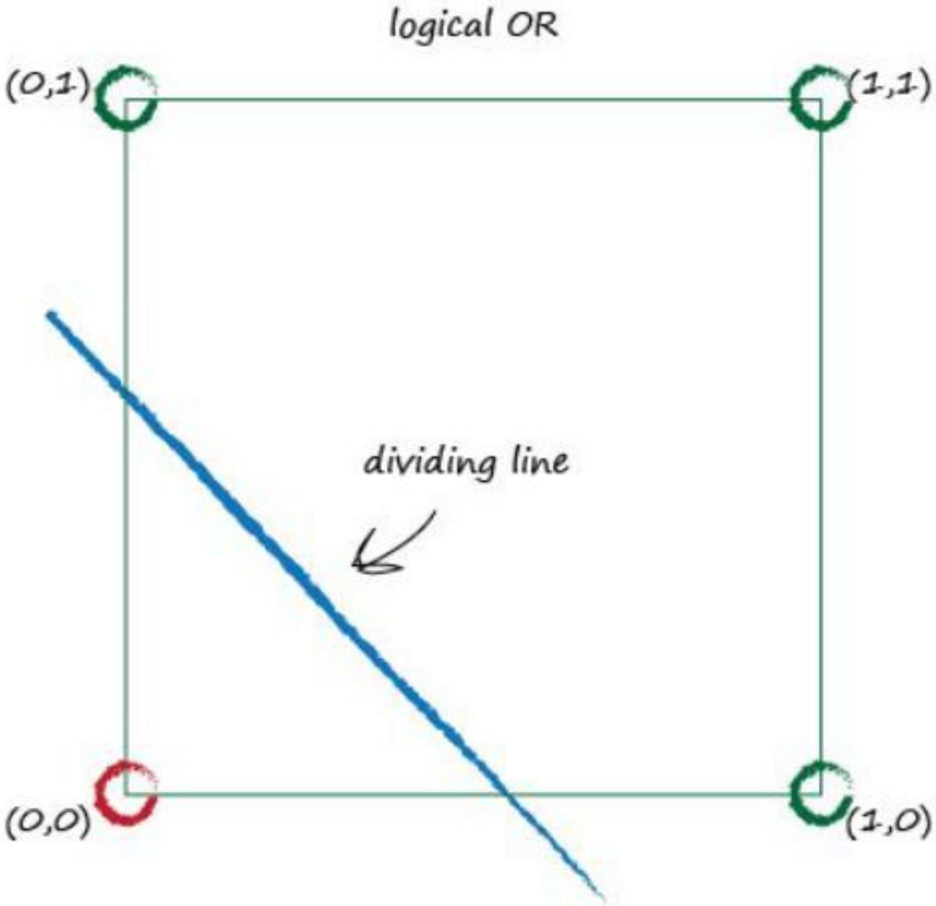

3
2
1
length
second moderated refinement
y = (1.6042) x
first moderated refinement
y = (0.3083) x
initial
y = (0.25) x
1
2
3
width

Sometimes One Classifier Is Not Enough

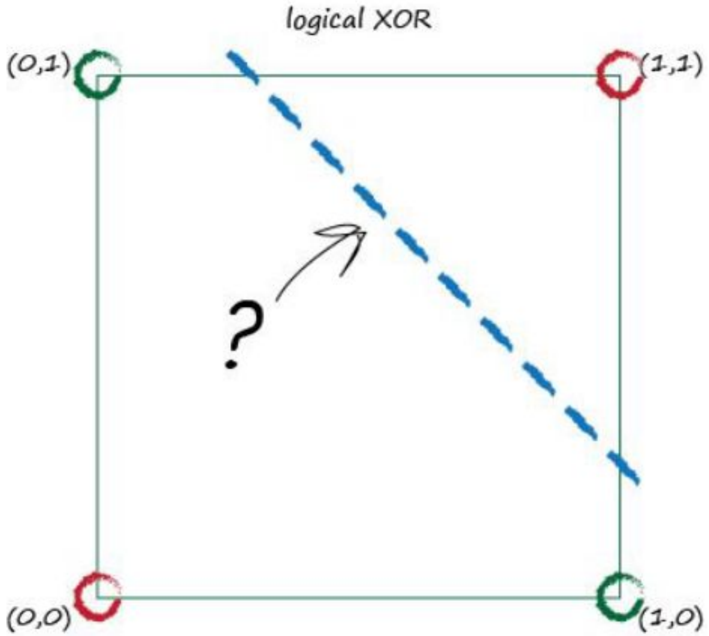


Input A	Input B	Logical AND	Logical OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1




logical OR
(0,1)
(1,1)
dividing line
(0,0)
(1,0)

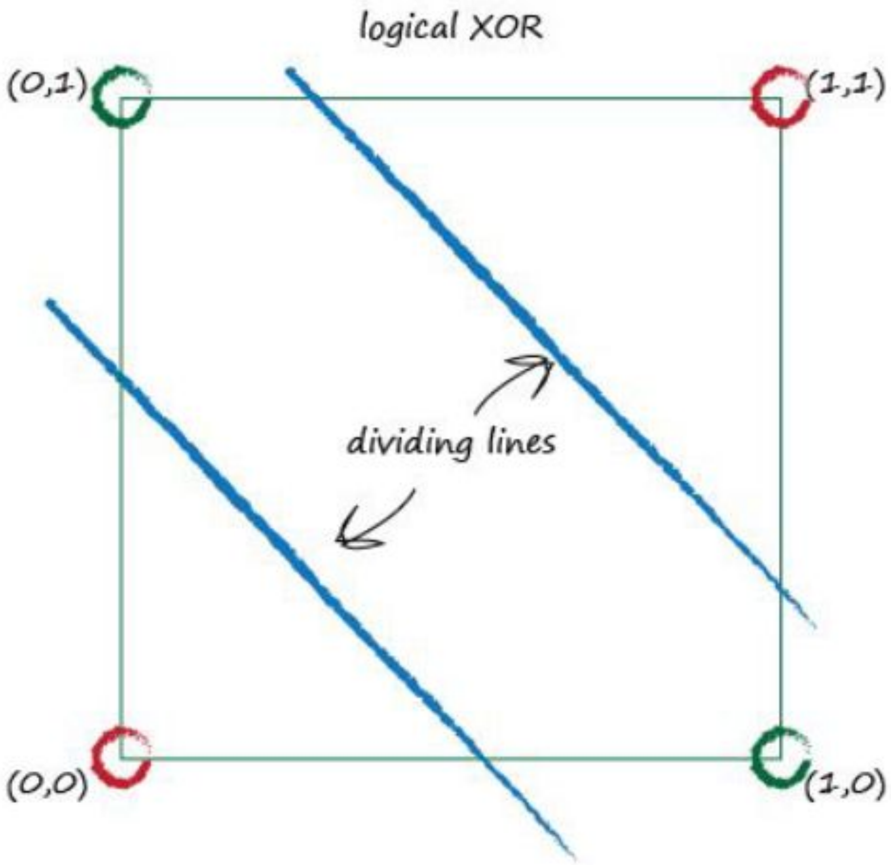
Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0



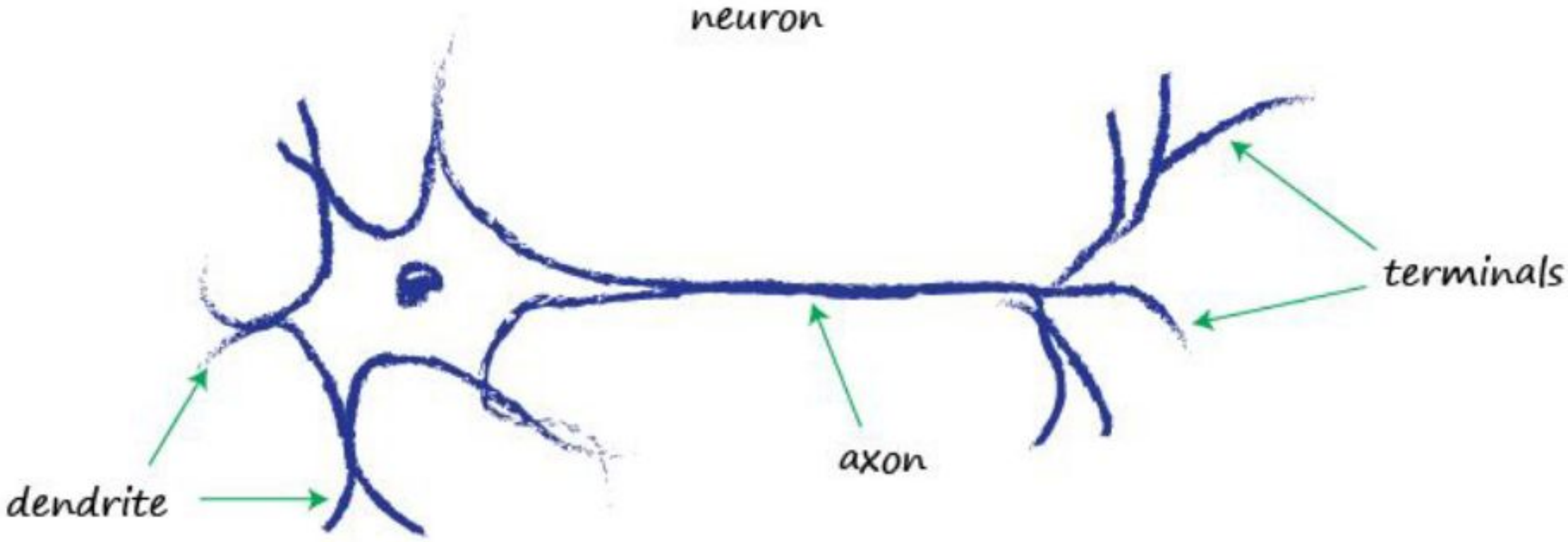
So we need a fix.

Luckily the fix is easy.

multiple classifiers working together



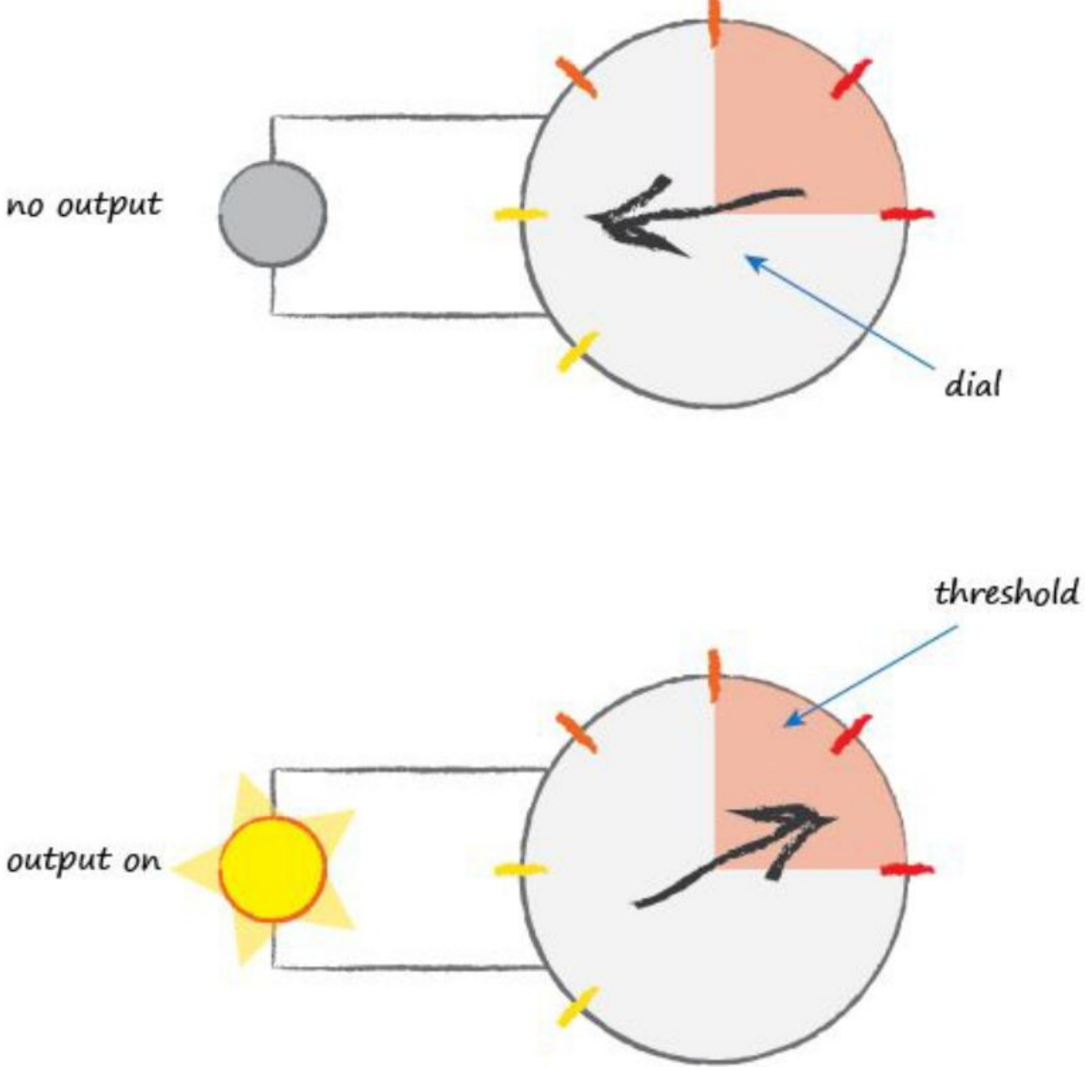
Neurons, Nature’s Computing Machines



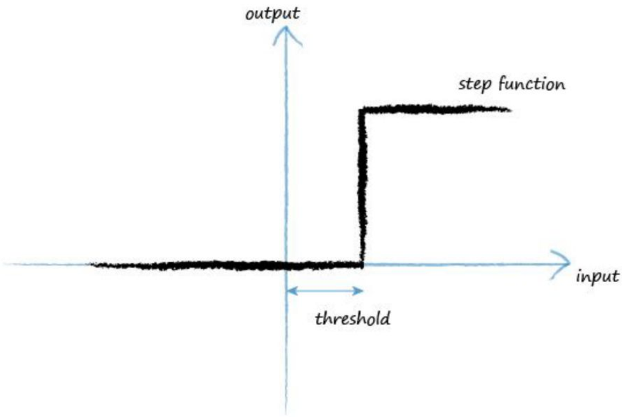
It takes an electric input, and pops out another electrical signal. This looks exactly like the classifying or predicting machines we looked at earlier, which took an input, did some processing, and popped out an output.

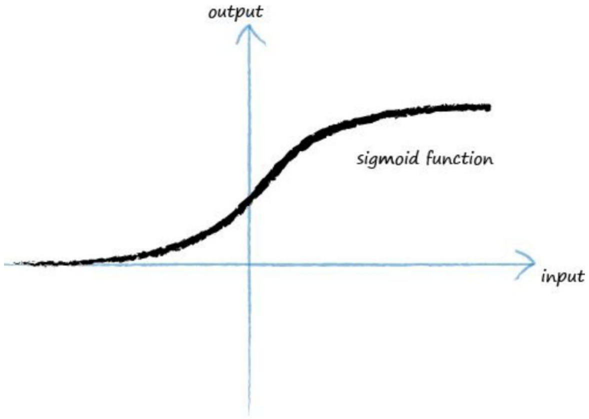
So could we represent neurons as linear functions, just like we did before?

A biological neuron doesn’t produce an output


no output
dial
threshold
output on

- A function that takes the input signal and generates an output signal, but takes into account some kind of threshold is called an **activation function**.
- Mathematically, there are many such activation functions that could achieve this effect. A simple step function could do this:

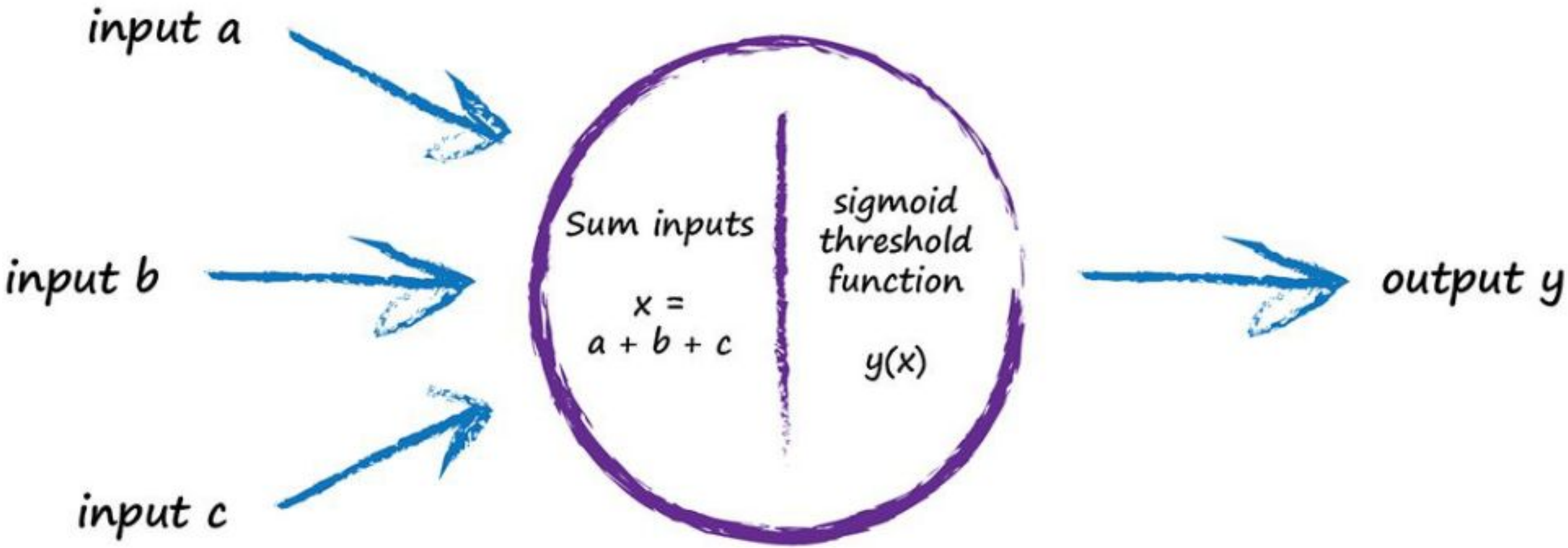




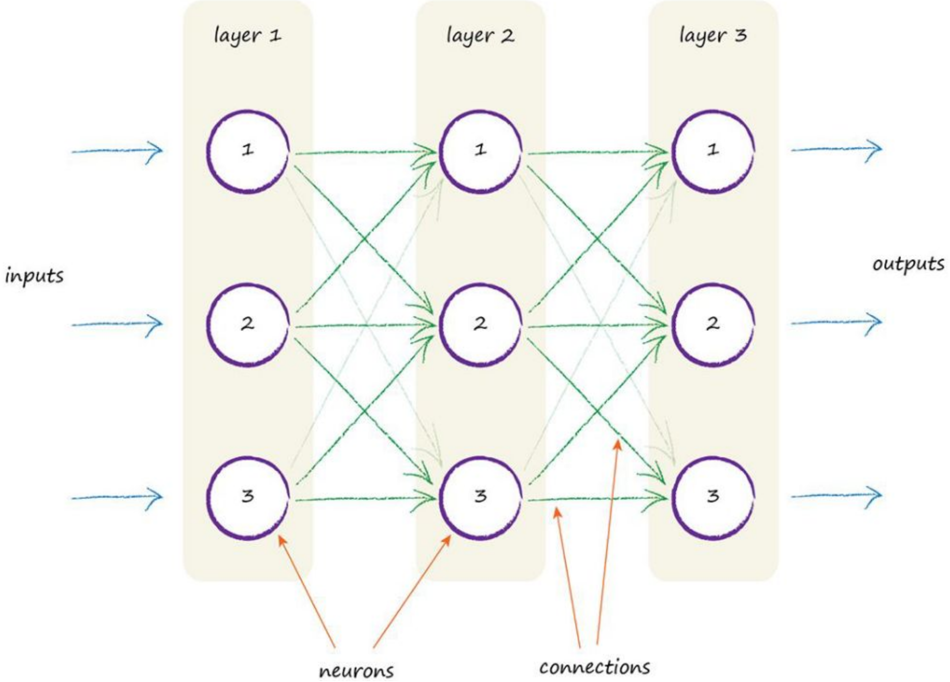
The sigmoid function, sometimes also called the logistic function, is

$$y = \frac{1}{1 + e^{-x}}$$

Idea of combining inputs

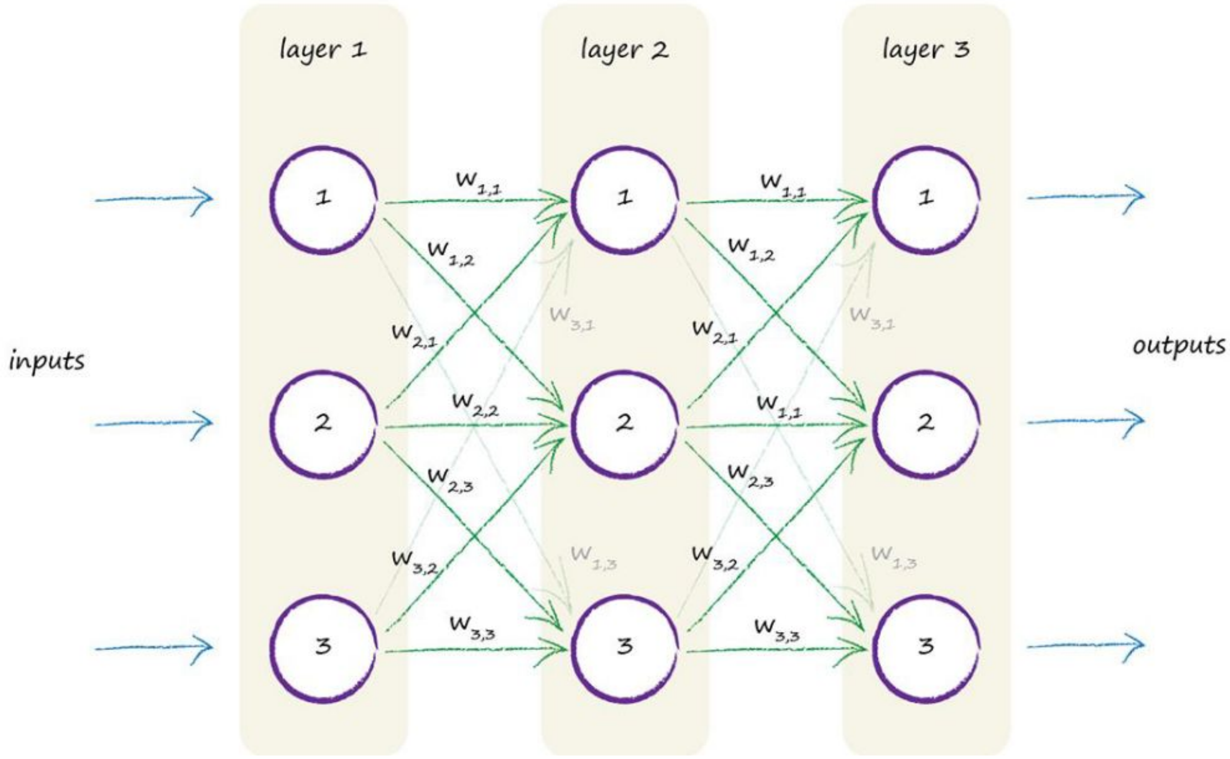


- What part of this cool looking architecture does the **learning**?
- What do we **adjust in response** to training examples?
- Is there a **parameter** that we can refine like the slope of the linear classifiers we looked at earlier?



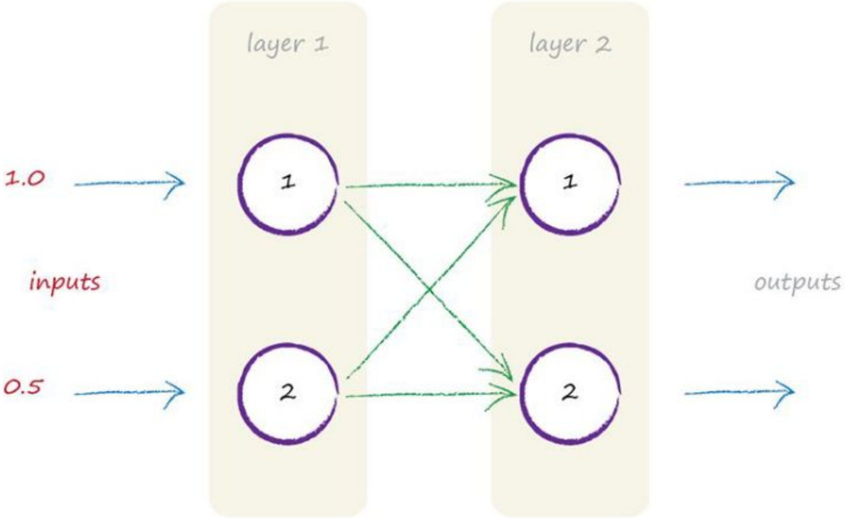
- The most obvious thing is to adjust the strength of the connections between nodes.
- Within a node, we could have adjusted the summation of the inputs, or we could have adjusted the shape of the sigmoid threshold function, but that’s more complicated than simply adjusting the strength of the connections between the nodes.

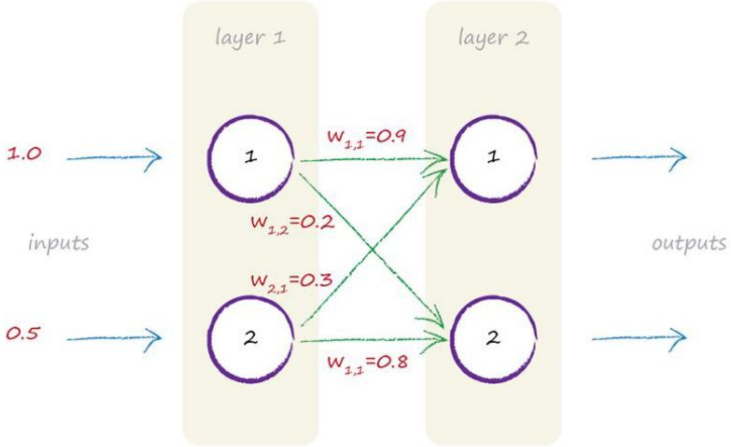
The following diagram again shows the connected nodes, but this time a weight is shown associated with each connection. A low weight will de emphasise a signal, and a high weight will amplify it.



- You might reasonably challenge this design and ask yourself why each node should connect to every other node in the previous and next layer.
- The learning process will de emphasise those few extra connections if they aren’t actually needed.
- What do we mean by this? It means that as the network learns to improve its outputs by refining the link weights inside the network, some weights become zero or close to zero. Zero, or almost zero, weights means those links don’t contribute to the network because signals don’t pass.
- A zero weight means the signals are multiplied by zero, which results in zero, so the link is effectively broken.

Following Signals Through A Neural Network





Let’s start calculating!

For each node in this layer we need to work out the combined input. Remember that sigmoid function, ?

So let’s first focus on node 1 in the layer 2.

The combined moderated input is:

x = (output from first node * link weight) + (output from second node * link weight)

x = (1.0 * 0.9) + (0.5 * 0.3)

x = 0.9 + 0.15

x = 1.05

We can now, finally, calculate that node’s output using the activation function

$$y = 1/(1 + e\ {-x})$$

The answer is

$$y = 1 / (1 + 0.3499) = 1 / 1.3499.$$

$$y = 0.7408.$$

Let’s do the calculation again with the remaining node which is node 2 in the second layer. The combined moderated input x is:

x = (output from first node * link weight) + (output from second node * link weight)

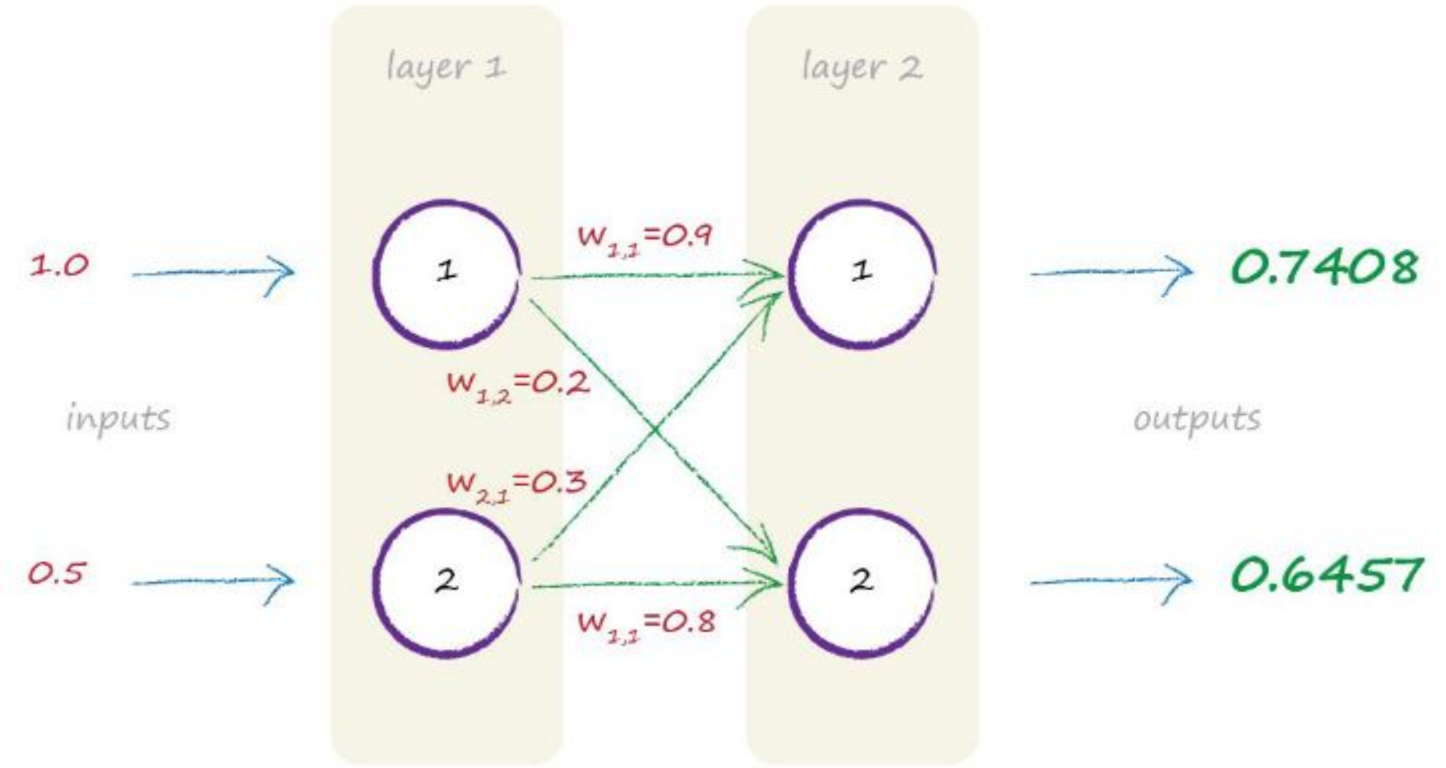
x = (1.0 * 0.2) + (0.5 * 0.8)

x = 0.2 + 0.4

x = 0.6

So now we have x, we can calculate the node’s output using the sigmoid activation function y = 1/(1 + 0.5488) = 1/(1.5488). So y = 0.6457.

The following diagram shows the network's outputs we've just calculated:



I wouldn’t want to do the calculations for a larger network by hand at all!

Look what happens if we replace the letters with words that are more meaningful to our neural networks. The second matrix is a two by one matrix, but the multiplication approach is the same.

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} \mathit{input\_1} \\ \mathit{input\_2} \end{pmatrix} = \begin{pmatrix} (\mathit{input\_1} * w_{1,1}) + (\mathit{input\_2} * w_{2,1}) \\ (\mathit{input\_1} * w_{1,2}) + (\mathit{input\_2} * w_{2,2}) \end{pmatrix}$$

The first matrix contains the weights between nodes of two layers. The second matrix contains the signals of the first input layer. The answer we get by multiplying these two matrices is the combined moderated signal into the nodes of the second layer

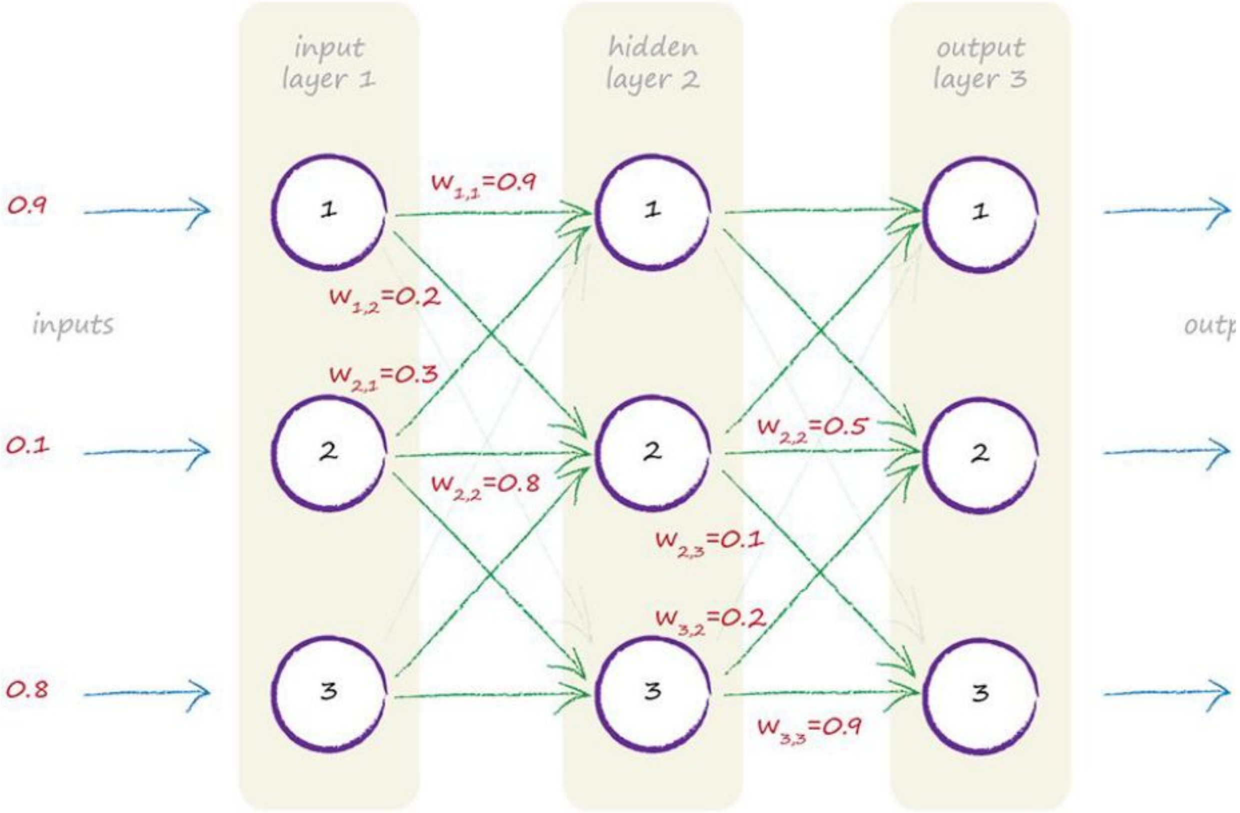
Because we can express all the calculations that go into working out the combined moderated signal, x, into each node of the second layer using matrix multiplication. And this can be expressed as concisely as:

$X = W \cdot I$

What about the activation function? the final output from the second layer is:

$O = \text{sigmoid}\ (X)$

Let us Start with 3 Layer NN



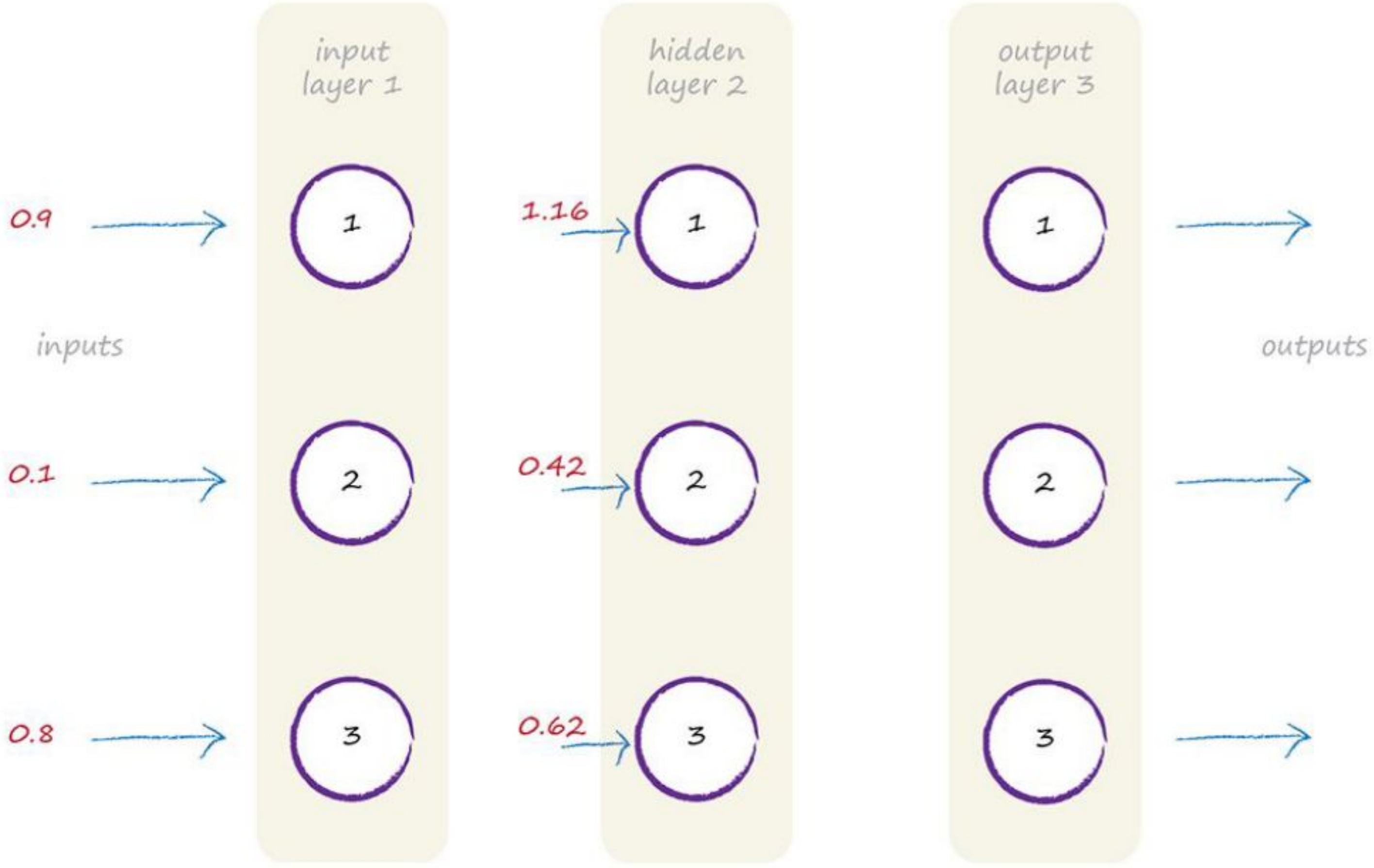
$$I = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix} \quad W_{input\_hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix}$$

$$W_{hidden\_output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

$$X_{hidden} = W_{input\_hidden} \cdot I$$

$$X_{hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

$$X_{hidden} = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

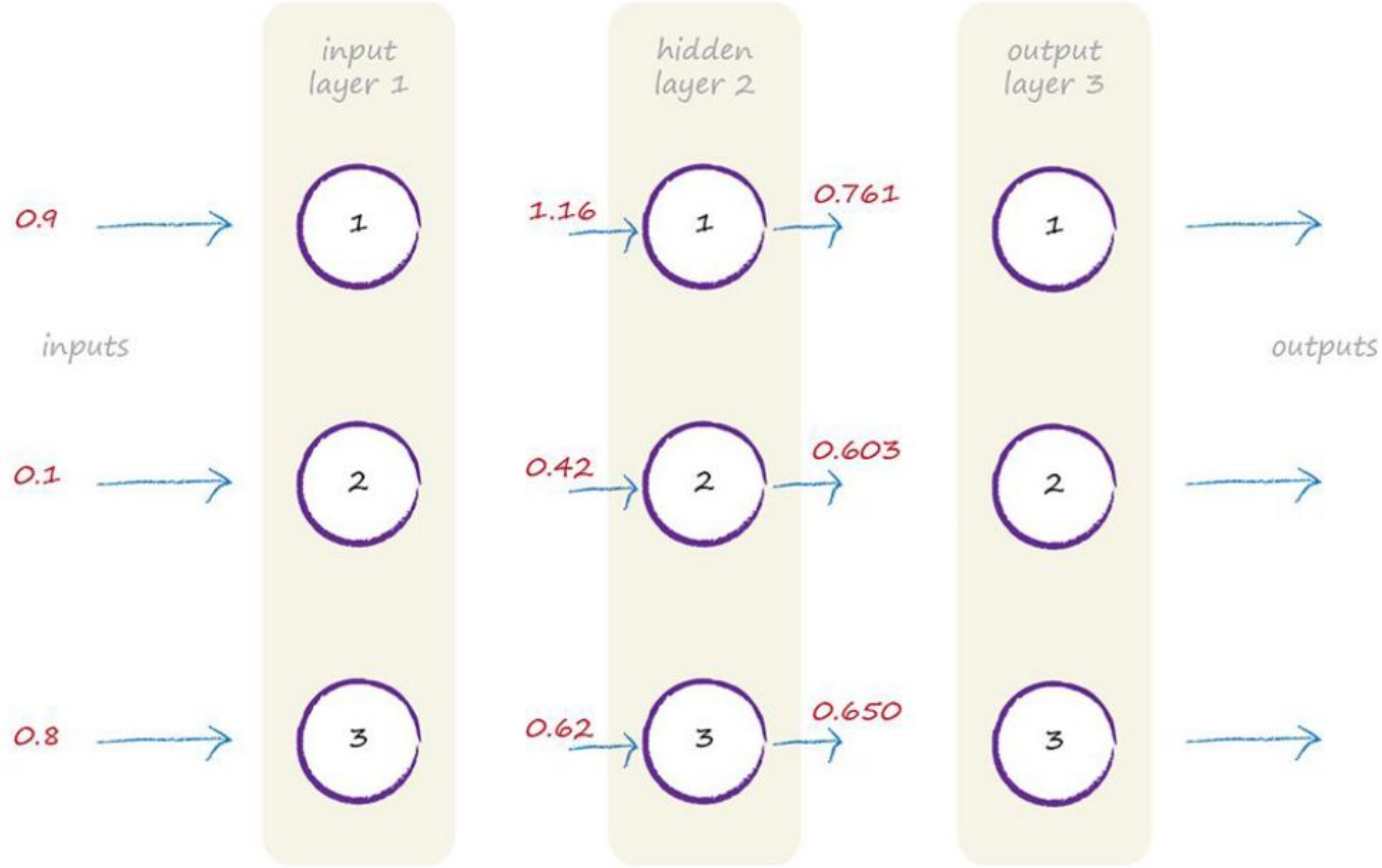

input
layer 1
hidden
layer 2
output
layer 3
0.9
0.1
0.8
inputs
1
2
3
1.16
0.42
0.62
1
2
3
1
2
3
outputs

The sigmoid function is applied to each element in Xhidden to produce the matrix which has the output of the middle hidden layer.

$$O_{hidden} = \text{signmoid}\begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$O_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

You can also see that all the values are between 0 and 1, because this sigmoid doesn’t produce values outside that range.

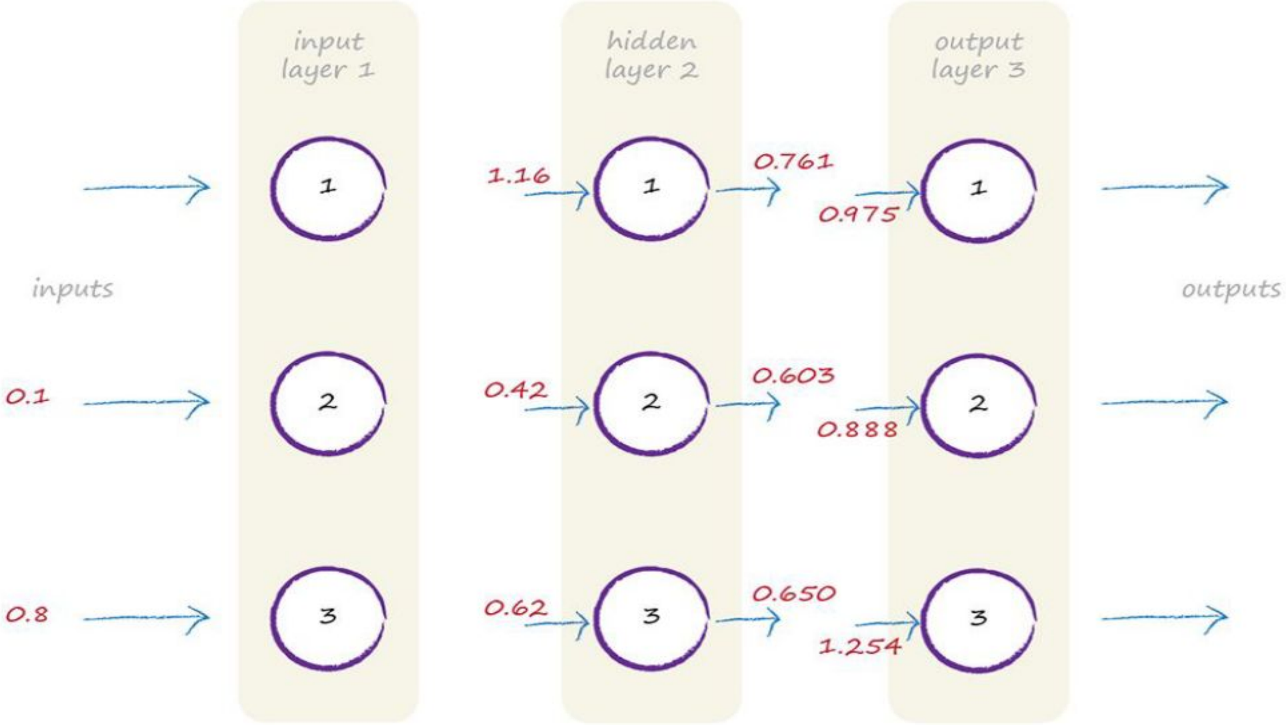

input layer 1
hidden layer 2
output layer 3
inputs
outputs
0.9
0.1
0.8
1
2
3
1.16
0.42
0.62
1
2
3
0.761
0.603
0.650
1
2
3

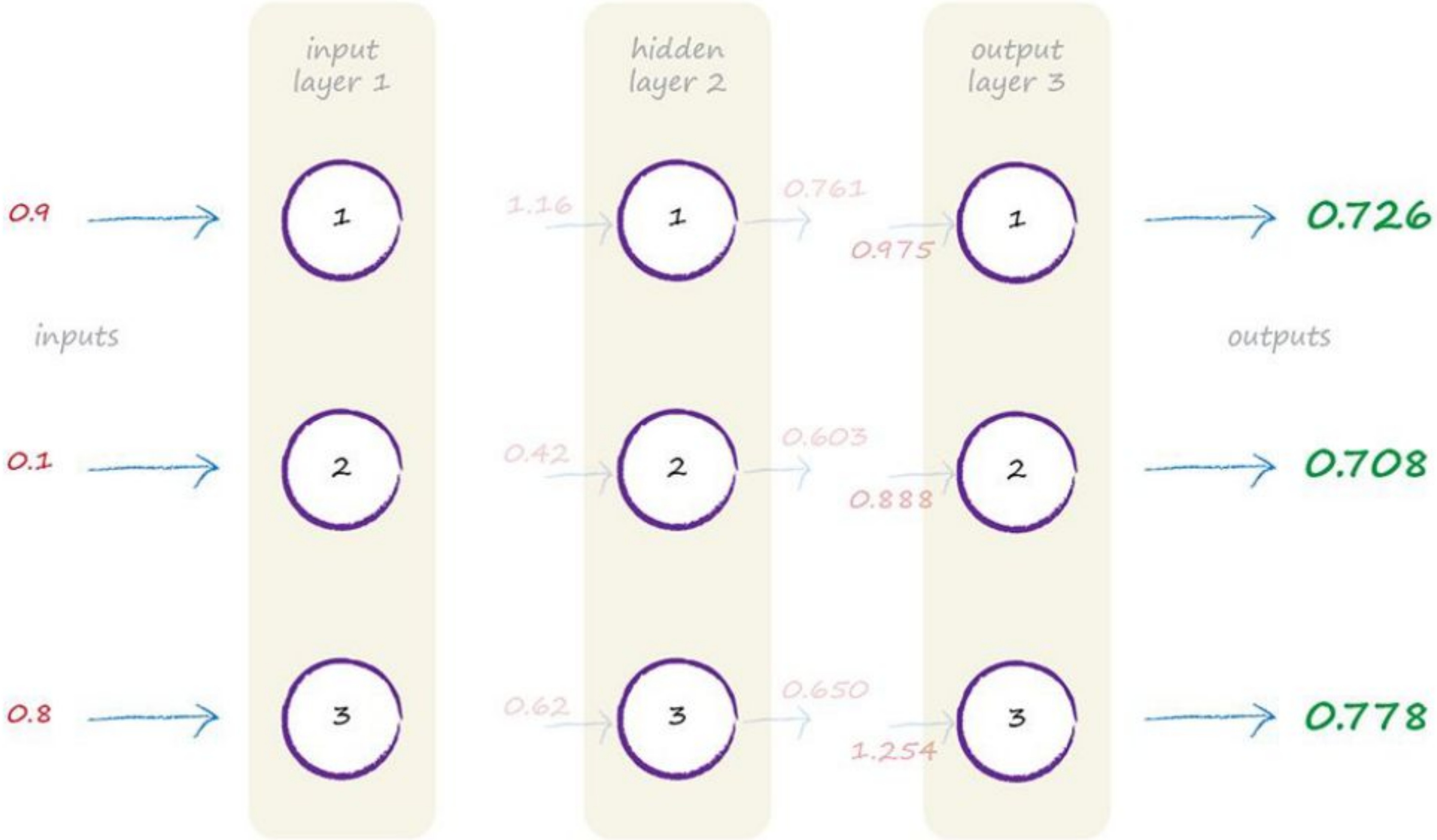
$$X_{output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

$$X_{output} = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$O_{output} = \text{signmoid}\begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$O_{output} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

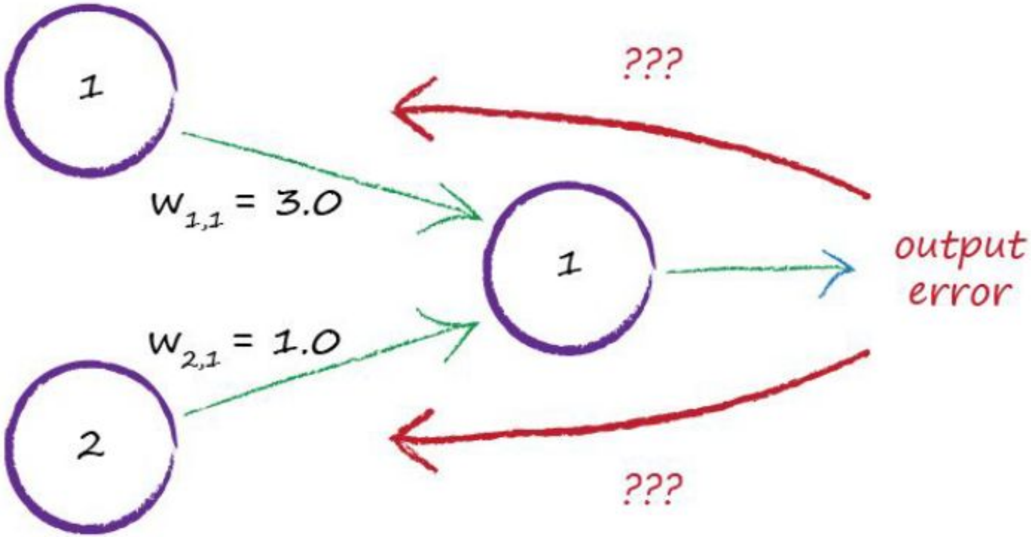


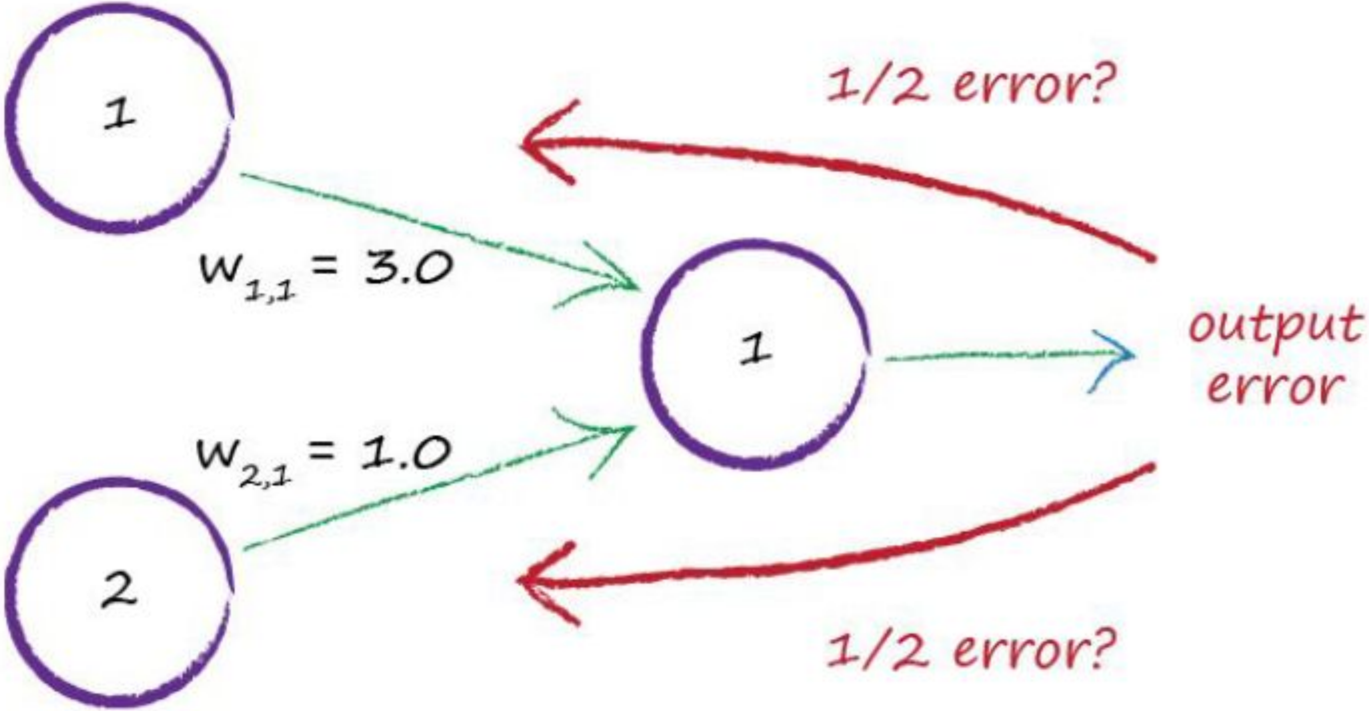

input
layer 1
hidden
layer 2
output
layer 3
0.9
0.1
0.8
inputs
1
2
3
1.16
0.42
0.62
1
2
3
0.761
0.603
0.650
0.975
0.888
1.254
1
2
3
0.726
0.708
0.778
outputs

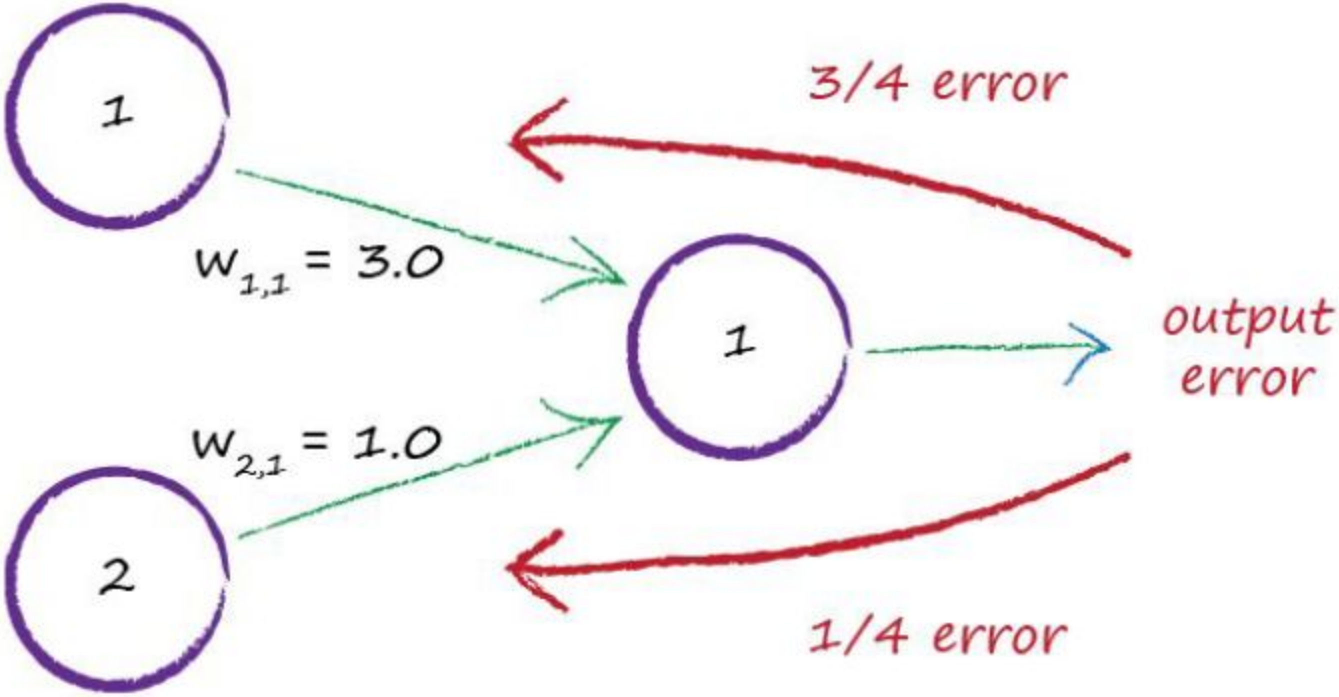
Learning Weights From More Than One Node

- The next step is to use the output from the neural network and compare it with the training example to work out an error.
- We need to use that error to refine the neural network itself so that it improves its outputs.
- We used the error, the difference between what the node produced as an answer and what we know the answer should be, to guide that refinement.
- That turned out to be quite easy because the relationship between the error and the necessary slope adjustment was very simple to work out.

How do we update link weights when more than one node contributes to an output and its error? The following illustrates this problem.




1
1/2 error?
w_{1,1} = 3.0
1
output
error
w_{2,1} = 1.0
2
1/2 error?

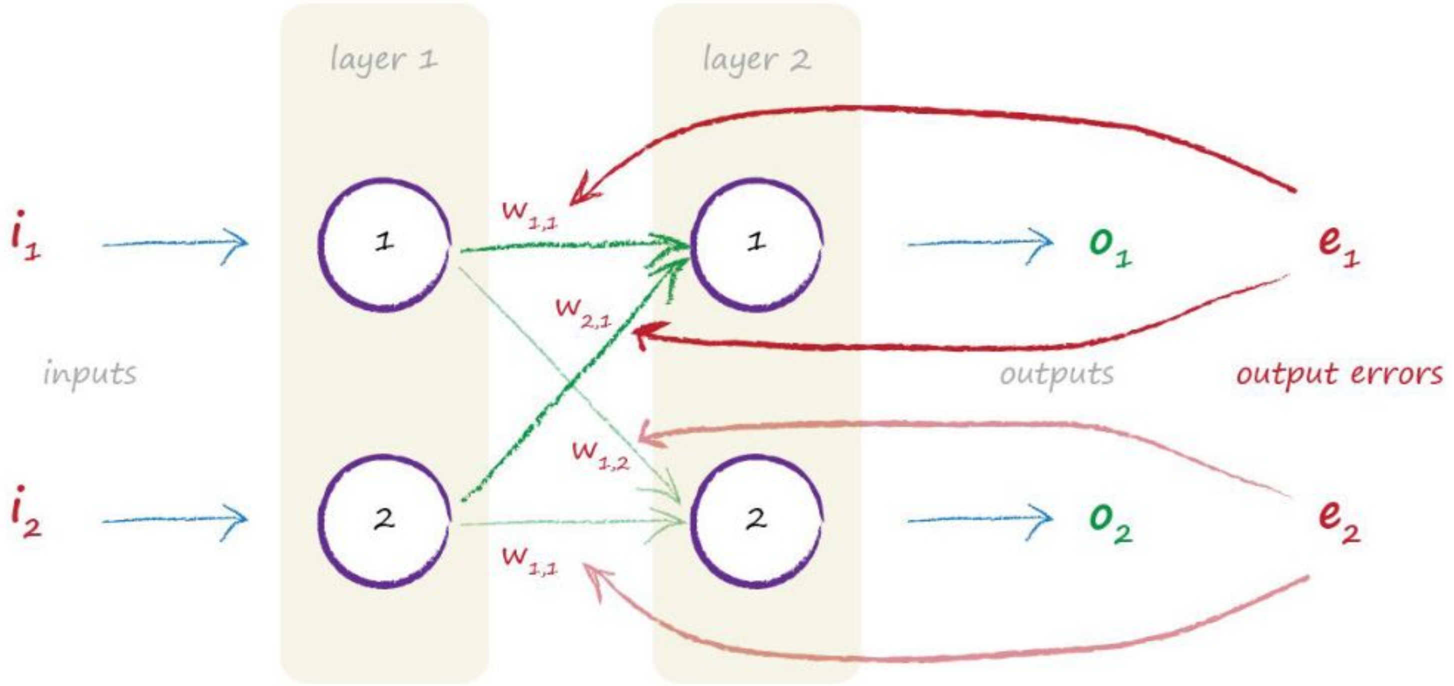

1
2
1
$w_{1,1} = 3.0$
$w_{2,1} = 1.0$
3/4 error
1/4 error
output
error

We can extend this same idea to many more nodes.

If we had 100 nodes connected to anoutput node, we’d split the error across the 100 connections to that output node in proportion to each link’s contribution to the error, indicated by the size of the link’s weight.

You can see that we’re using the weights in two ways.

Backpropagating Errors From More Output Nodes



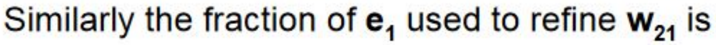
You can see from the diagram that the error e1 is split in proportion to the connected links, which have weights w11 and w21.

Similarly, e2 would be split in proportionate to weights w21 and w22.

Let’s write out what these splits are, so we’re not in any doubt. The error e1 is used to inform the refinement of both weights w11 and w21. It is split so that the fraction of e1 used to update w11 is

These fractions might look a bit puzzling, so let’s illustrate what they do. Behind all these symbols is the every simple idea the the error e1 is split to give more to the weight that is larger, and less to the weight that is smaller.

$$\frac{w_{11}}{w_{11} + w_{21}}$$



Similarly the fraction of $\mathbf{e}_1$ used to refine $\mathbf{w}_{21}$ is

$$\frac{w_{21}}{w_{11} + w_{21}}$$

Backpropagating Errors with Matrix Multiplication

Try to learn this

How Do We Actually Update Weights?

Issues:

- These nodes aren’t simple linear classifiers.
- 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 3 layers and 3 neurons in each layer, like we had above.

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 3 layer neural network with 3 nodes in each layer.

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

Solution!!!!

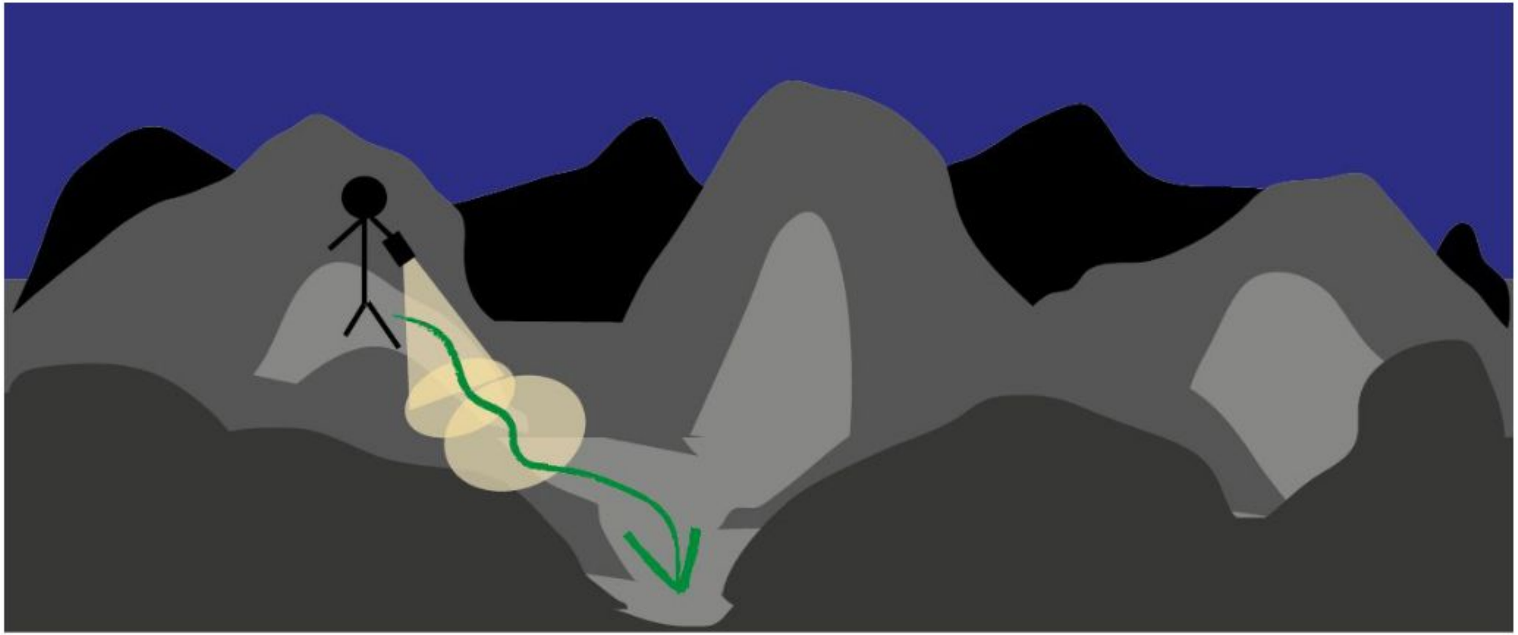
Instead of trying to be too clever, we could just simply try random combinations of weights until we find a good one? The approach is called a brute force method.

So how do we solve such an apparently hard problem?

- The mathematical expressions showing how all the weights result in a neural network's output are too complex to easily untangle.
- The weight combinations are too many to test one by one to find the best.
- The training data might not be sufficient to properly teach a network. The training data might have errors so our assumption that it is the perfect truth, something to learn from, is then flawed.
- The network itself might not have enough layers or nodes to model the right solution to the problem.

What this means is we must take an approach that is realistic, and recognises these limitations.

Let’s illustrate what we mean by this



The mathematical version of this approach is called gradient descent, and you can see why.

After you’ve taken a step, you look again at the surrounding area to see which direction takes you closer to your objective, and then you step again in that direction.

You keep doing this until you’re happy you’ve arrived at the bottom. The gradient refers to the slope of the ground. Youstep in the direction where the slope is steepest downwards.Now imagine that complex landscape is a mathematical function. What this gradient descent method gives us is an ability to find the minimum without actually having to understand that complex function enough to work it out mathematically.

- It might not give us the exact answer because we’re using steps to approach an answer, improving our position bit by bit.
- But that is better than not having an answer at all.
- Anyway, we can keep refining the answer with ever smaller steps towards the actual minimum, until we’re happy with the accuracy we’ve achieved.

What’s the link between this really cool gradient descent method and neural networks? Well, if

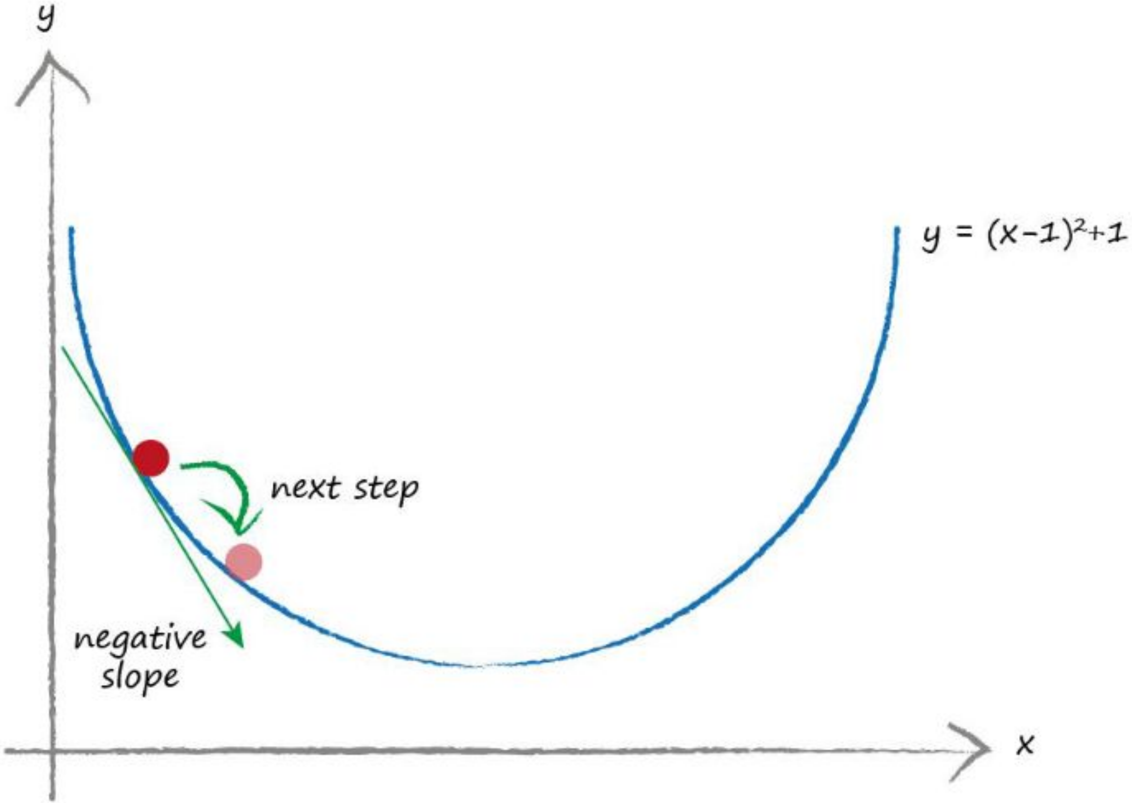
the complex difficult function is the error of the network, then going downhill to find the minimum

means we’re minimising the error. We’re improving the network’s output. That’s what we want!

What’s the link between this really cool gradient descent method and neural networks?

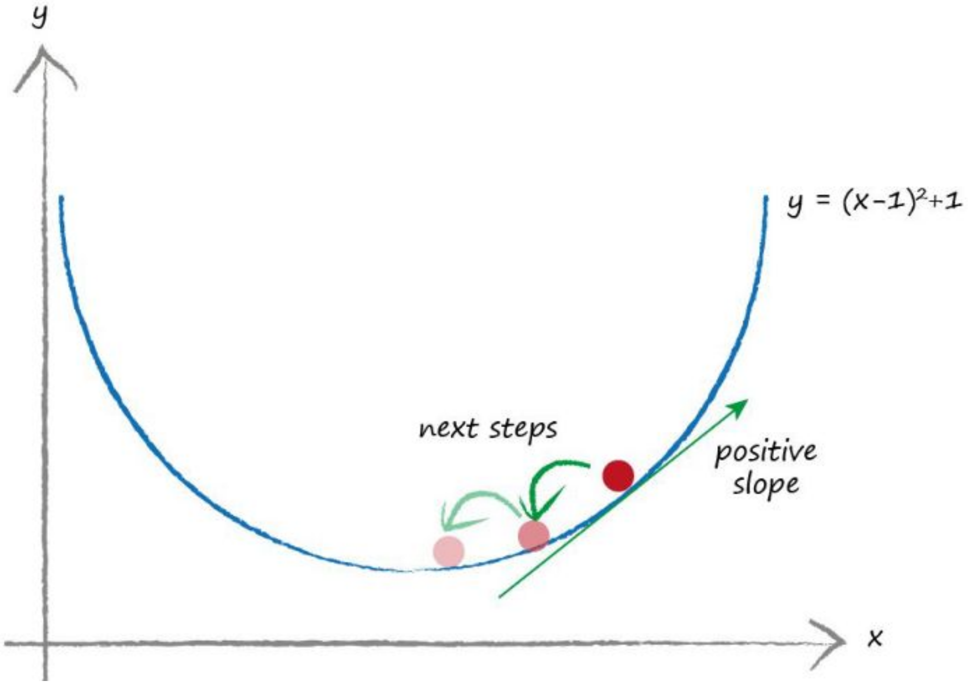
- If the complex difficult function is the error of the network, then going downhill to find the minimum means we’re minimising the error.
- We’re improving the network’s output.

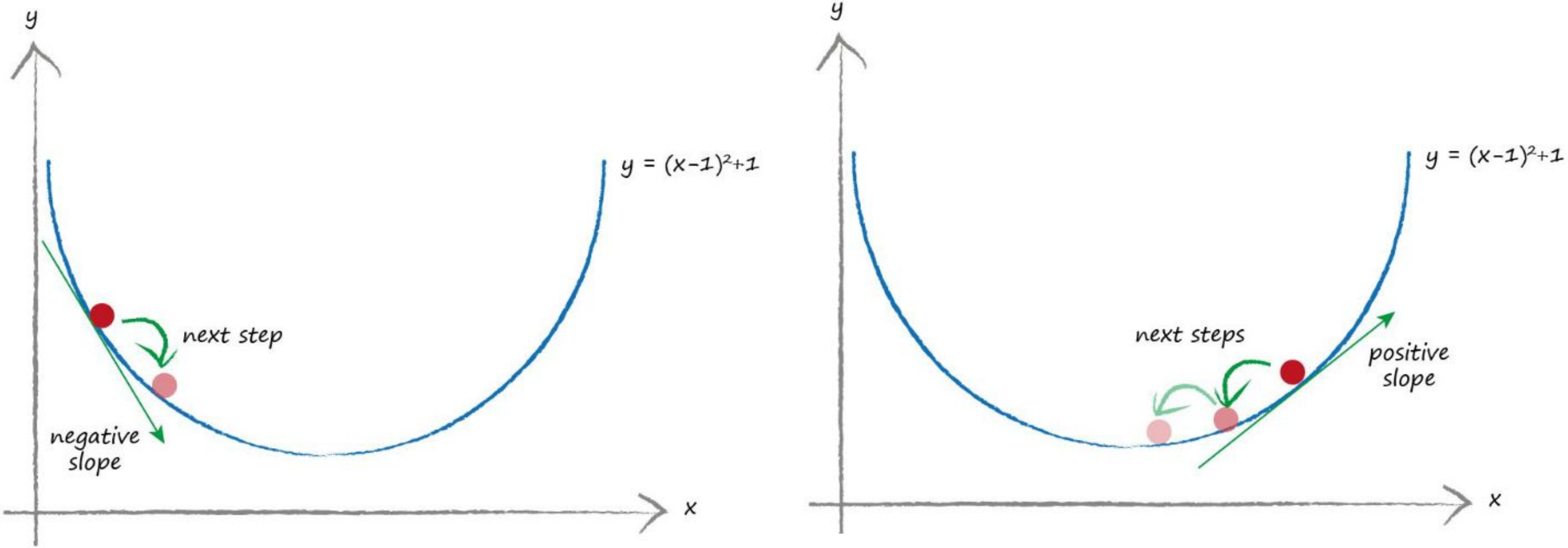
That’s what we want!


y
$y = (x-1)^2+1$
next step
negative slope
x

Rule!!!

To do gradient descent we have to start somewhere.




y
y = (x-1)²+1
next step
negative slope
x
y
y = (x-1)²+1
next steps
positive slope
x

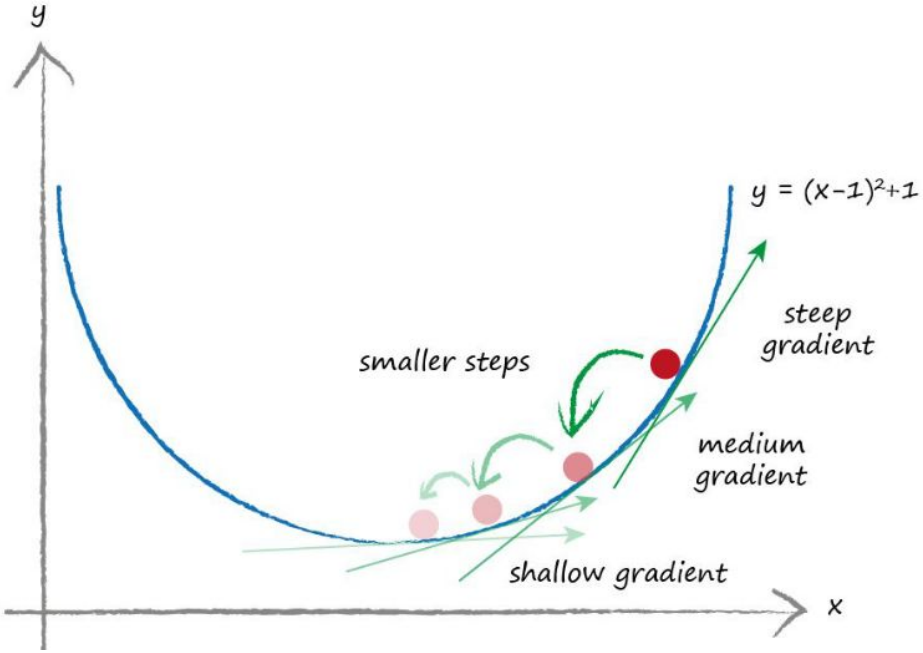
Gradient is a measure of how steep a slope is.

- **The greater the gradient the steeper a slope is.**
- **The smaller the gradient the shallower a slope is.**

In other terms:

A shallow gradient indicates **a slow rate of reaction** - the steeper the gradient, the faster the rate of reaction.

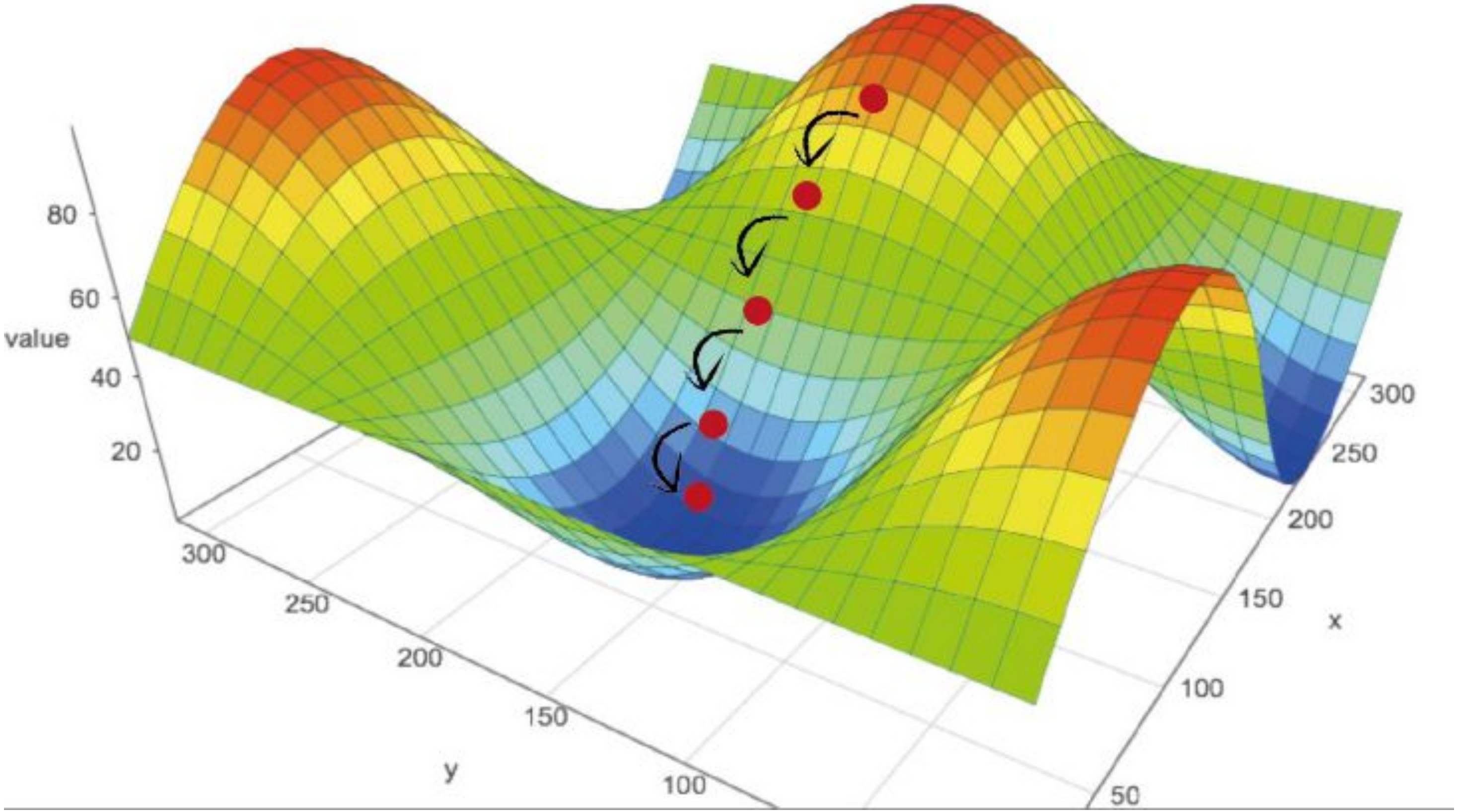
A positive gradient means we reduce x. A negative gradient means we increase x.



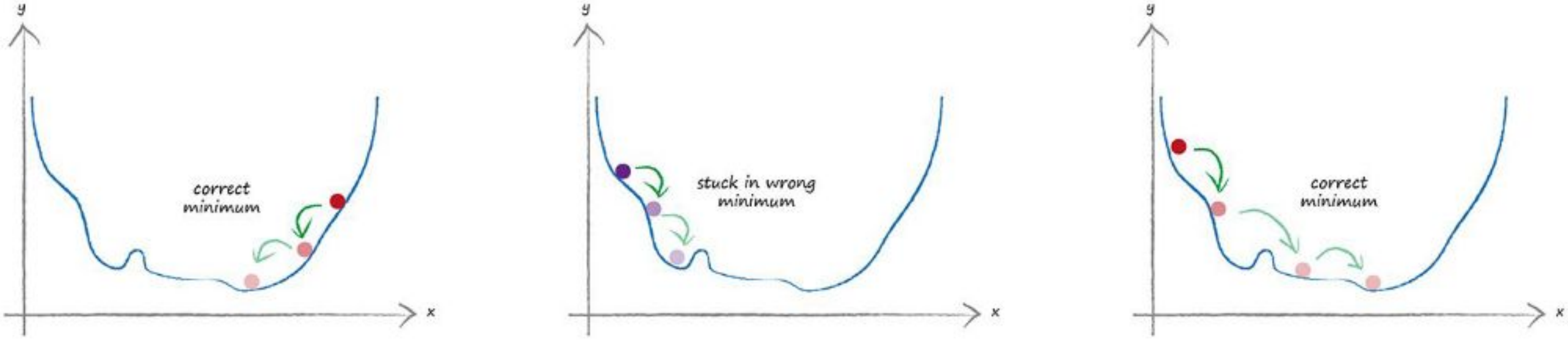
This method really shines when we have functions of many parameters.

So not just y depending on x, but maybe y depending on a, b, c, d, e and f. Remember the output function, and therefore the error function, of a neural network depends on many many weight parameters.

The following again illustrates gradient descent but with a slightly more complex function that depends on 2 parameters.


80
60
value
40
20
300
250
200
150
y
100
300
250
200
150
x
100
50

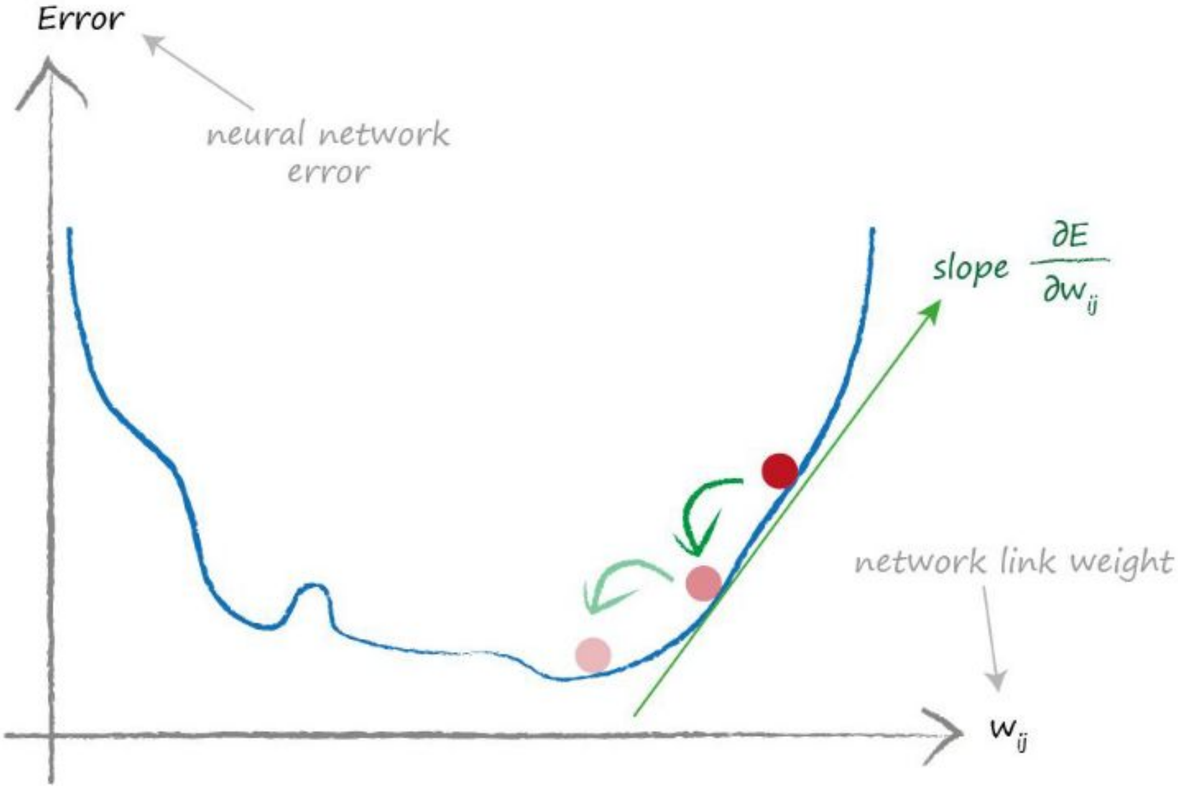
To avoid ending up in the wrong valley, or function minimum, we train neural

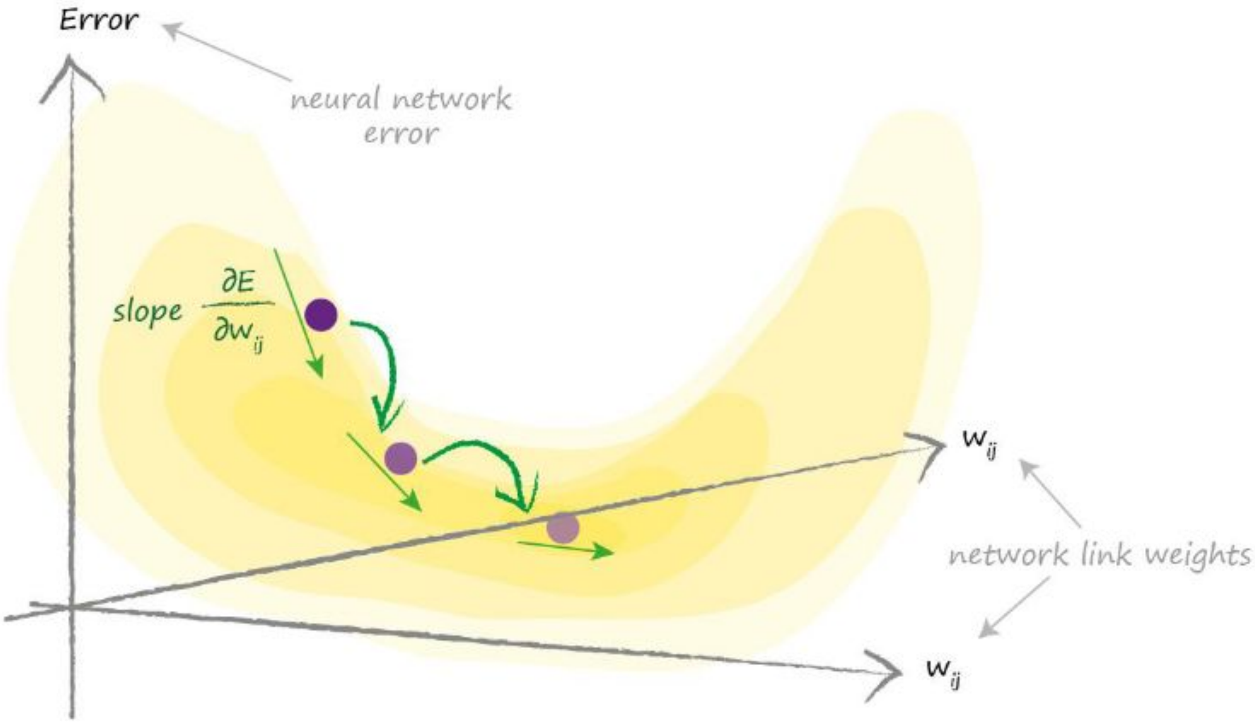


Network Output	Target Output	Error (target - actual)	Error |target - actual|	Error $(\text{target - actual})^2$
0.4	0.5	0.1	0.1	0.01
0.8	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

We have a fourth option?

- Yes, you can construct all kind of complicated and interesting **cost functions.**
- Some don’t work well at all, some work well for particular kinds of problems, and some do work but aren’t worth the extra complexity.


Error
neural network error
slope $\frac{\partial E}{\partial w_{ij}}$
network link weight
w_{ij}

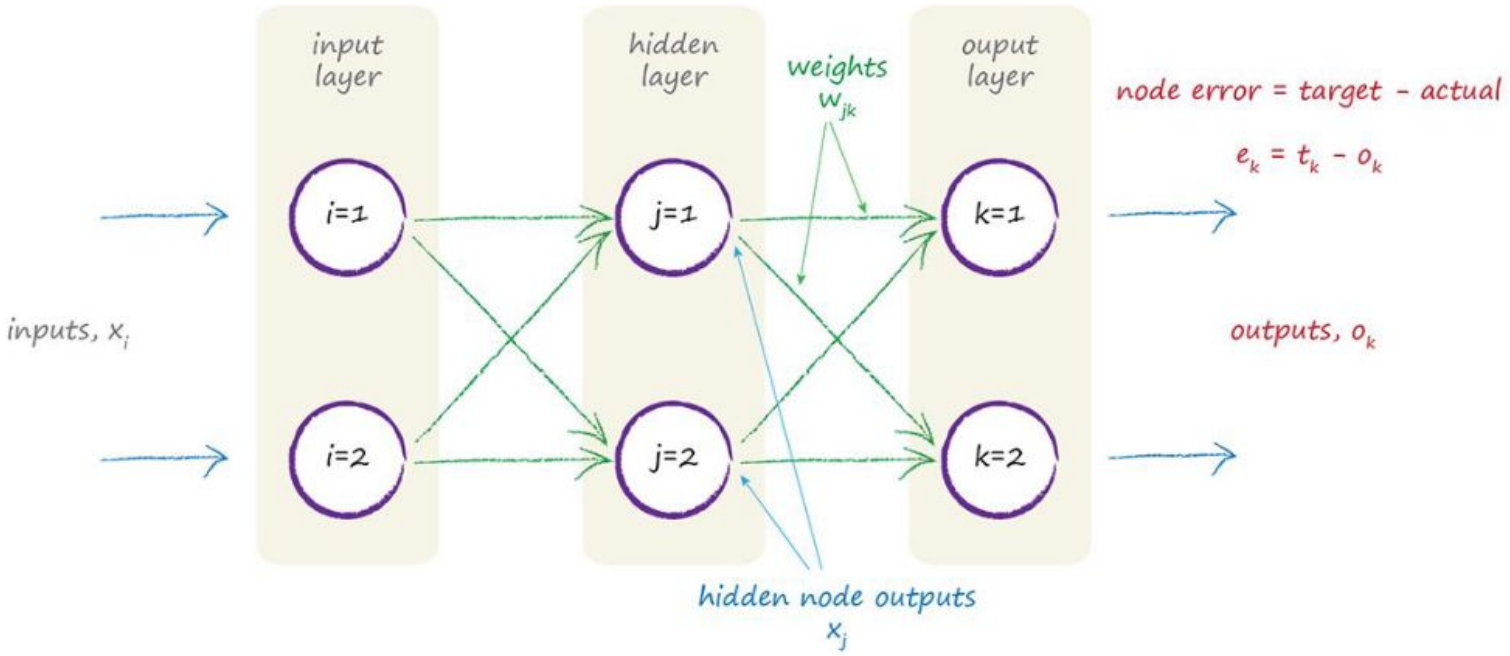

Error
neural network error
slope $\frac{\partial E}{\partial w_{ij}}$
w_{ij}
network link weights
w_{ij}

It’s harder to visualise 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.

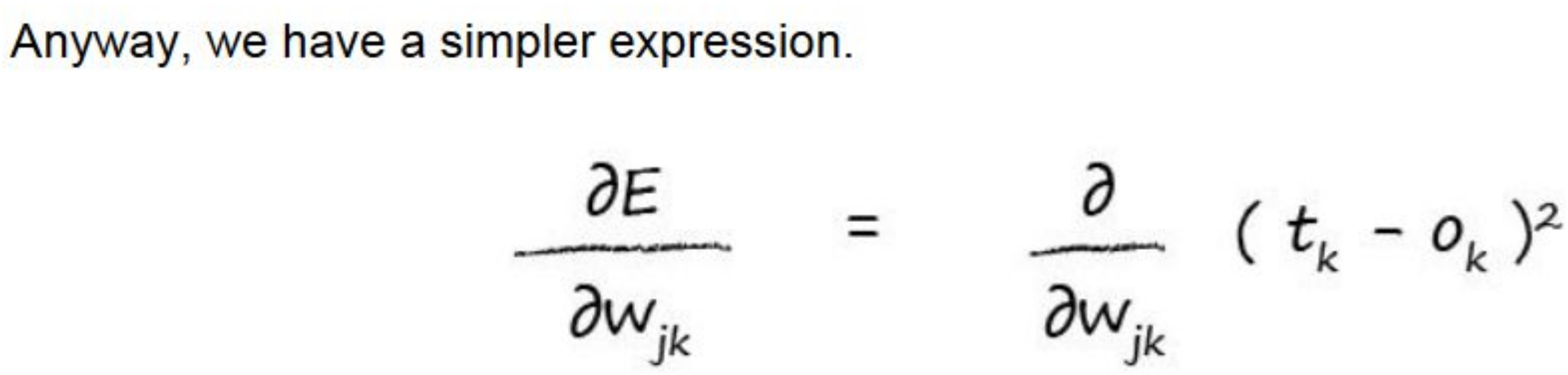
$$\frac{\partial E}{\partial w_{jk}}$$

A derivative is a continuous description of how a function changes with small changes in one or multiple variables.


input layer
hidden layer
weights w_{jk}
ouput layer
node error = target - actual
$e_k = t_k - o_k$
i=1
i=2
j=1
j=2
k=1
k=2
inputs, x_i
outputs, o_k
hidden node outputs x_j

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.

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



Anyway, we have a simpler expression.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

We’ll use the chain rule to break apart this differentiation task into more manageable pieces.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

That ok is the output of the node k which, if you remember, is the sigmoid function applied to the weighted sum of the connected incoming signals. So let’s write that out to make it clear

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \, sigmoid(\Sigma_j \, w_{jk} \cdot o_j)$$

That oj is the output from the previous hidden layer node, not the output from the final layer ok.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \,.\, \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j \right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j \right)\right) \,.\, \frac{\partial}{\partial w_{jk}}\left(\Sigma_j w_{jk} \cdot o_j \right)$$

$$= -2(t_k - o_k) \,.\, \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j \right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j \right)\right) \,.\, o_j$$

As we know: $\frac{\partial}{\partial x} \text{sigmoid}(x) = \text{sigmoid}(x) \left(1 - \text{sigmoid}(x) \right)$

The final answer we’ve been working towards, the one that describes the slope of the error function so we can adjust the weight wjk.

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\right) \cdot o_j$$

- The first part which was the (target actual) error now becomes the recombined back propagated error out of the hidden nodes, just as we saw above. Let’s call that ej.
- ● The sigmoid parts can stay the same, but the sum expressions inside refer to the preceding layers, so the sum is over all the inputs moderated by the weights into a hidden node j. We could call this ij.
- ● The last part is now the output of the first layer of nodes oi, which happen to be the input signals.

So the second part of the final answer we’ve been striving towards is as follows, the slope of the error function for the weights between the input and hidden layers.

$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot \text{sigmoid}(\Sigma_i w_{ij} \cdot o_i)(1 - \text{sigmoid}(\Sigma_i w_{ij} \cdot o_i)) \cdot o_i$$

We’ve now got all these crucial magic expressions for the slope, we can use them to update the weights after each training example as follows

- Remember the weights are changed in a direction opposite to the gradient, as we saw clearly in the diagrams earlier.
- We also moderate the change by using a learning factor, which we can tune for a particular problem.
- We saw this too when we developed linear classifiers as a way to avoid being pulled too far wrong by bad training examples, but also to ensure the weights don’t bounce around a minimum by constantly overshooting it.

Let’s say this in mathematical form.

$$\text{new } w_{jk} = \text{old } w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

The updated weight wjk is the old weight adjusted by the negative of the error slope we just worked out. It’s negative because we want to increase the weight if we have a positive slope, and decrease it if we have a negative slope, as we saw earlier. The symbol alpha ƀ, is a factor which moderates the strength of these changes to make sure we don’t overshoot. It’s often called a learning rate.

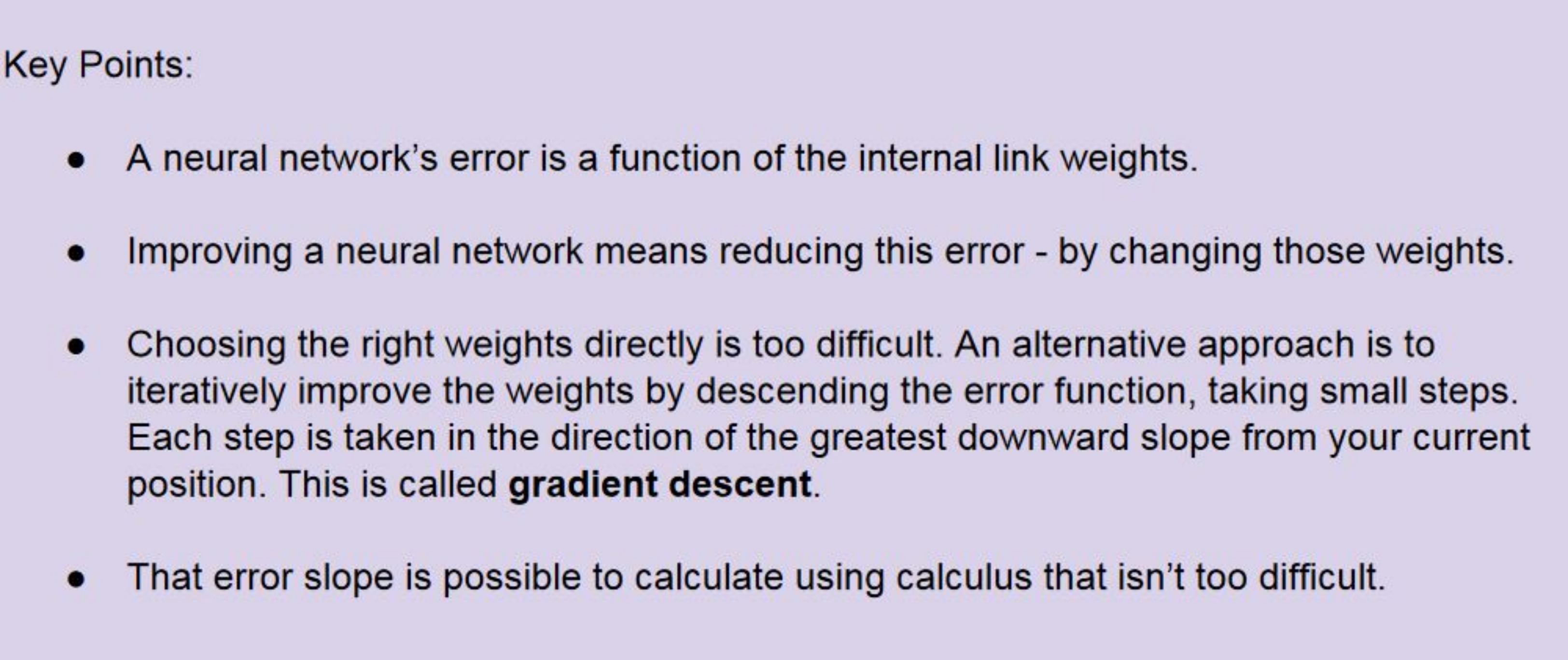
$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1 (1-S_1) \\ E_2 * S_2 (1-S_3) \\ E_k * S_k (1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

values from previous layer

values from next layer

So, the matrix form of these weight update matrices, is as follow, ready for us to implement in a computer programming language that can work with matrices efficiently.

$$\Delta W_{jk} = \alpha * E_k * sigmoid\,(O_k) * (1 - sigmoid\,(O_k)) \cdot O_j^T$$



Key Points:

- A neural network’s error is a function of the internal link weights.

- Improving a neural network means reducing this error - by changing those weights.

- Choosing the right weights directly is too difficult. An alternative approach is to iteratively improve the weights by descending the error function, taking small steps. Each step is taken in the direction of the greatest downward slope from your current position. This is called **gradient descent**.

- That error slope is possible to calculate using calculus that isn’t too difficult.