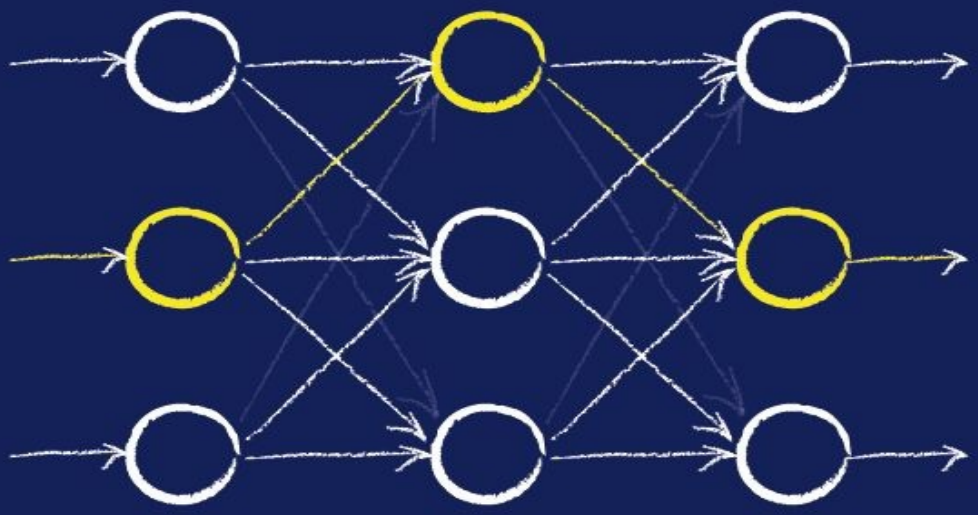
MAKE YOUR OWN NEURAL NETWORK



A gentle journey through the mathematics of neural networks, and making your own using the Python computer language.

TARIQ RASHID

Table of Contents

Prologue

The Search for Intelligent Machines

For thousands of years, we humans have tried to understand how our own intelligence works and replicate it in some kind of machine - thinking machines.

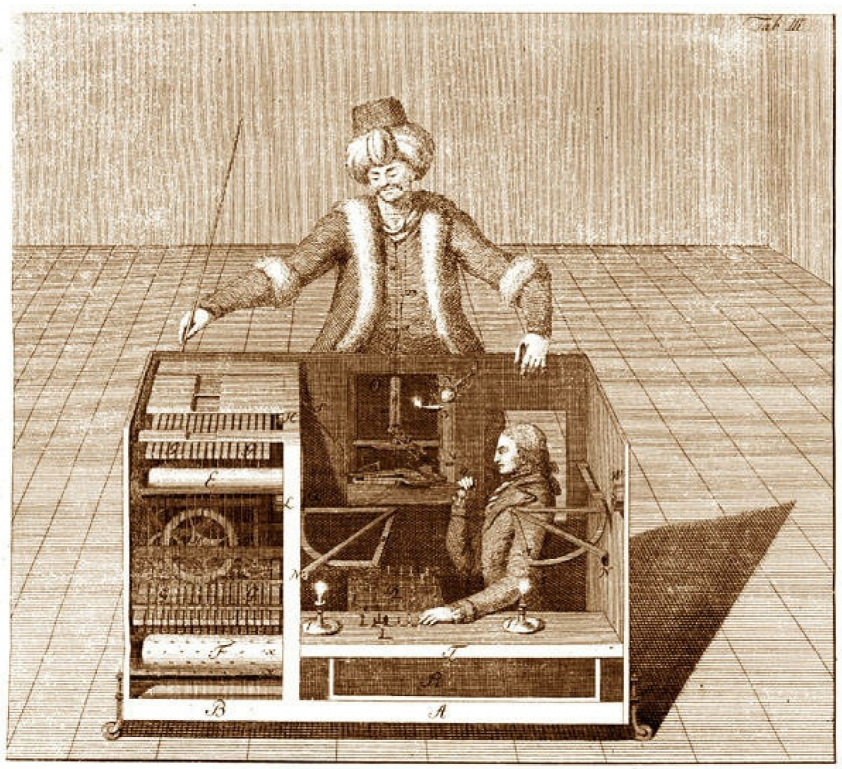
We've not been satisfied by mechanical or electronic machines helping us with simple tasks - flint sparking fires, pulleys lifting heavy rocks, and calculators doing arithmetic.

Instead, we want to automate more challenging and complex tasks like grouping similar photos, recognising diseased cells from healthy ones, and even putting up a decent game of chess. These tasks seem to require human intelligence, or at least a more mysterious deeper capability of the human mind not found in simple machines like calculators.

Machines with this human-like intelligence is such a seductive and powerful idea that our culture is full of fantasies, and fears, about it - the immensely capable but ultimately menacing HAL 9000 in Stanley Kubrick's *2001: A Space Odyssey*, the crazed action *Terminator* robots and the talking KITT car with a cool personality from the classic *Knight Rider* TV series.

When Gary Kasparov, the reigning world chess champion and grandmaster, was beaten by the IBM Deep Blue computer in 1977 we feared the potential of machine intelligence just as much as we celebrated that historic achievement.

So strong is our desire for intelligent machines that some have fallen for the temptation to cheat. The infamous mechanical Turk chess machine was merely a hidden person inside a cabinet!



A Nature Inspired New Golden Age

Optimism and ambition for artificial intelligence were flying high when the subject was formalised in the 1950s. Initial successes saw computers playing simple games and proving theorems. Some were convinced machines with human level intelligence would appear within a decade or so.

But artificial intelligence proved hard, and progress stalled. The 1970s saw a devastating academic challenge to the ambitions for artificial intelligence, followed by funding cuts and a loss of interest.

It seemed machines of cold hard logic, of absolute 1s and 0s, would never be able to achieve the nuanced organic, sometimes fuzzy, thought processes of biological brains.

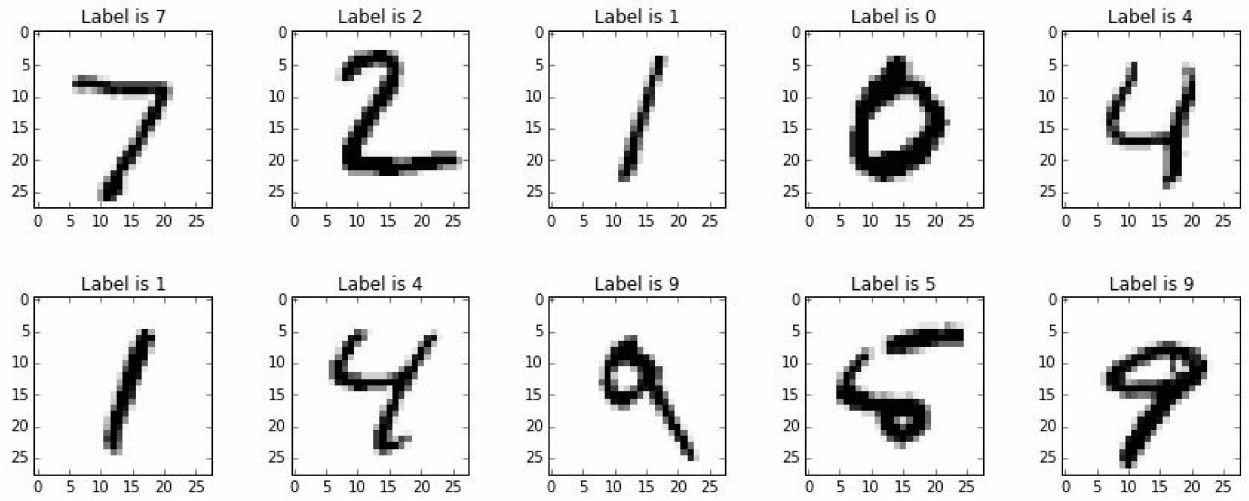
After a period of not much progress an incredibly powerful idea emerged to lift the search for machine intelligence out of its rut. Why not try to build artificial brains by copying how real biological brains worked? Real brains with neurons instead of logic gates, softer more organic reasoning instead of the cold hard, black and white, absolutist traditional algorithms.

Scientist were inspired by the apparent simplicity of a bee or pigeon's brain compared to the complex tasks they could do. Brains a fraction of a gram seemed able to do things like steer flight and adapt to wind, identify food and predators, and quickly decide whether to fight or escape. Surely computers, now with massive cheap resources, could mimic and improve on these brains? A bee has around 950,000 neurons - could today's computers with gigabytes and terabytes of resources outperform bees?

But with traditional approaches to solving problems - these computers with massive storage and superfast processors couldn't achieve what the relatively miniscule brains in birds and bees could do.

Neural networks emerged from this drive for biologically inspired intelligent computing - and went on to become one of the most powerful and useful methods in the field of artificial intelligence. Today, Google's Deepmind, which achieves fantastic things like learning to play video games by itself, and for the first time beating a world master at the incredibly rich game of Go, have neural networks at their foundation. Neural networks are already at the heart of everyday technology - like automatic car number plate recognition and decoding handwritten postcodes on your handwritten letters.

This guide is about neural networks, understanding how they work, and making your own neural network that can be trained to recognise human handwritten characters, a task that is very difficult with traditional approaches to computing.


Label is 7
Label is 2
Label is 1
Label is 0
Label is 4
Label is 1
Label is 4
Label is 9
Label is 5
Label is 9

Introduction

Who is this book for?

This book is for anyone who wants to understand what neural network are. It's for anyone who wants to make and use their own. And it's for anyone who wants to appreciate the fairly easy but exciting mathematical ideas that are at the core of how they work.

This guide is not aimed at experts in mathematics or computer science. You won't need any special knowledge or mathematical ability beyond school maths.

If you can add, multiply, subtract and divide then you can make your own neural network. The most difficult thing we'll use is gradient calculus - but even that concept will be explained so that as many readers as possible can understand it.

Interested readers or students may wish to use this guide to go on further exciting excursions into artificial intelligence. Once you've grasped the basics of neural networks, you can apply the core ideas to many varied problems.

Teachers can use this guide as a particularly gentle explanation of neural networks and their implementation to enthuse and excite students making their very own learning artificial intelligence with only a few lines of programming language code. The code has been tested to with a Raspberry Pi, a small inexpensive computer very popular in schools and with young students.

I wish a guide like this had existed when I was a teenager struggling to work out how these powerful yet mysterious neural networks worked. I'd seen them in books, films and magazines, but at that time I could only find difficult academic texts aimed at people already expert in mathematics and its jargon.

All I wanted was for someone to explain it to me in a way that a moderately curious school student could understand. That's what this guide wants to do.

What will we do?

In this book we'll take a journey to making a neural network that can recognise human handwritten numbers.

We'll start with very simple predicting neurons, and gradually improve on them as we hit their limits. Along the way, we'll take short stops to learn about the few mathematical concepts that are needed to understand how neural networks learn and predict solutions to problems.

We'll journey through mathematical ideas like functions, simple linear classifiers, iterative refinement, matrix multiplication, gradient calculus, optimisation through gradient descent and even geometric rotations. But all of these will be explained in a really gentle clear way, and will assume absolutely no previous knowledge or expertise beyond simple school mathematics.

Once we've successfully made our first neural network, we'll take idea and run with it in different directions. For example, we'll use image processing to improve our machine learning without resorting to additional training data. We'll even peek inside the mind of a neural network to see if it reveals anything insightful - something not many guides show you how to do!

We'll also learn Python, an easy, useful and popular programming language, as we make our own neural network in gradual steps. Again, no previous programming experience will be assumed or needed.

How will we do it?

The primary aim of this guide is to open up the concepts behind neural networks to as many people as possible. This means we'll always start an idea somewhere really comfortable and familiar. We'll then take small easy steps, building up from that safe place to get to where we have just enough understanding to appreciate something really cool or exciting about the neural networks.

To keep things as accessible as possible we'll resist the temptation to discuss anything that is more than strictly required to make your own neural network. There will be interesting context and tangents that some readers will appreciate, and if this is you, you're encouraged to research them more widely.

This guide won't look at all the possible optimisations and refinements to neural networks. There are many, but they would be a distraction from the core purpose here - to introduce the essential ideas in as easy and uncluttered was as possible.

This guide is intentionally split into three sections:

- In **part 1** we'll gently work through the mathematical ideas at work inside simple neural networks. We'll deliberately not introduce any computer programming to avoid being distracted from the core ideas.

- In **part 2** we'll learn just enough Python to implement our own neutral network. We'll train it to recognise human handwritten numbers, and we'll test its performance.

- In **part 3**, we'll go further than is necessary to understand simple neural networks, just to have some fun. We'll try ideas to further improve our neural network's performance, and we'll also have a look inside a trained network to see if we can understand what it has learned, and how it decides on its answers.

And don't worry, all the software tools we'll use will be **free** and **open source** so you won't have to pay to use them. And you don't need an expensive computer to make your own neural network. All the code in this guide has been tested to work on a very inexpensive £5 / $4 Raspberry Pi Zero, and there's a section at the end explaining how to get your Raspberry Pi ready.

Author's Note

I will have failed if I haven't given you a sense of the true excitement and surprises in mathematics and computer science.

I will have failed if I haven't shown you how school level mathematics and simple computer recipes can be incredibly powerful - by making our own artificial intelligence mimicking the learning ability of human brains.

I will have failed if I haven't given you the confidence and desire to explore further the incredibly rich field of artificial intelligence.

I welcome feedback to improve this guide. Please get in touch at makeyourownneuralnetwork at gmail dot com, or on twitter @myoneuralnet.

You will also find discussions about the topics covered here at http://makeyourownneuralnetwork.blogspot.co.uk. There will be an errata of corrections there too.

Part 1 - How They Work

“Take inspiration from all the small things around you.”

Easy for Me, Hard for You

Computers are nothing more than calculators at heart. They are very very fast at doing arithmetic.

This is great for doing tasks that match what a calculator does - summing numbers to work out sales, applying percentages to work out tax, plotting graphs of existing data.

Even watching catch-up TV or streaming music through your computer doesn't involve much more than the computer executing simple arithmetic instructions again and again. It may surprise you but reconstructing a video frame from the ones and zeros that are piped across the internet to your computer is done using arithmetic not much more complex than the sums we did at school.

Adding up numbers really quickly - thousands, or even millions, a second - may be impressive but it isn't artificial intelligence. A human may find it hard to do large sums very quickly but the process of doing it doesn't require much intelligence at all. It simply requires an ability to follow very basic instructions, and this is what the electronics inside a computer does.

Now let's flips things and turn the tables on computers!

Look at the following images and see if you can recognise what they contain:

You and I can look at a picture with human faces, a cat, or a tree, and recognise it. In fact we can do it rather quickly, and to a very high degree of accuracy. We don't often get it wrong.

We can process the quite large amount of information that the images contain, and very successfully process it to recognise what's in the image. This kind of task isn't easy for computers - in fact it's incredibly difficult.

Problem	Computer	Human
Multiply thousands of large numbers quickly	Easy	Hard
Find faces in a photo of a crowd of people	Hard	Easy

We suspect image recognition needs human intelligence - something machines lack, however complex and powerful we build them, because they're not human.

But it is exactly these kinds of problems that we want computers to get better at solving - because they're fast and don't get tired. And it these kinds of hard problems that artificial intelligence is all about.

Of course computers will always be made of electronics and so the task of artificial intelligence is to find new kinds of recipes, or **algorithms**, which work in new ways to try to solve these kinds of harder problem. Even if not perfectly well, but well enough to give an impression of a human like intelligence at work.

Key Points:

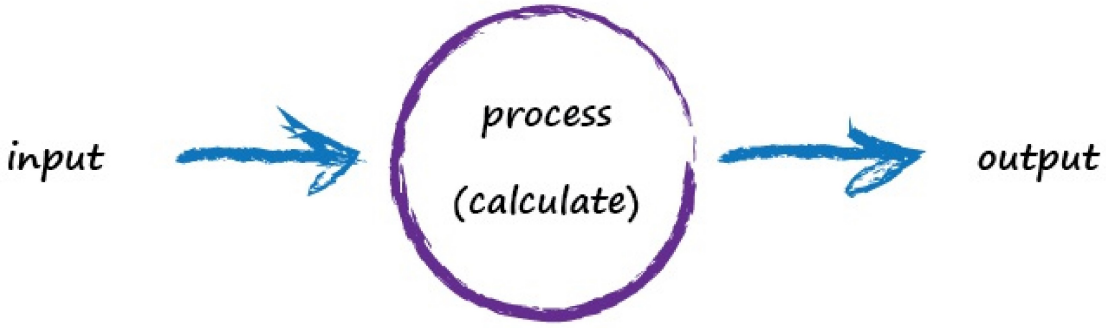
- Some tasks are easy for traditional computers, but hard for humans. For example, multiplying millions of pairs of numbers.
- On the other hand, some tasks are hard for traditional computers, but easy for humans. For example, recognising faces in a photo of a crowd.

A Simple Predicting Machine

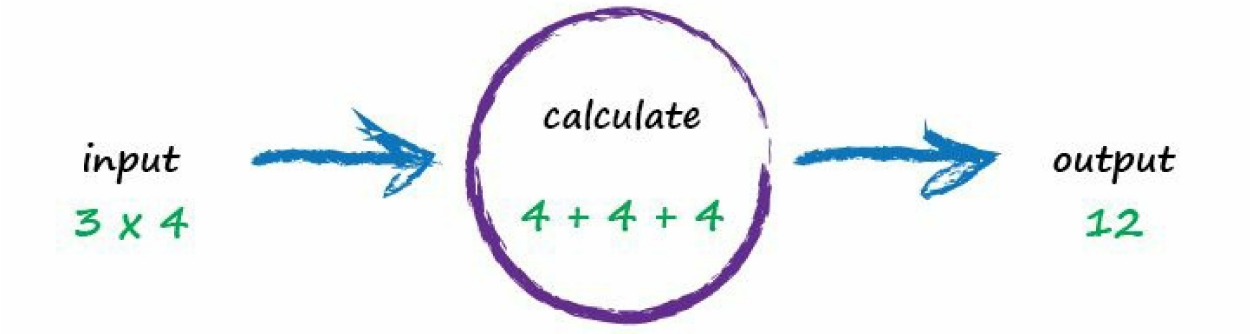
Let's start super simple and build up from there.

Imagine a basic machine that takes a question, does some "thinking" and pushes out an answer. Just like the example above with ourselves taking input through our eyes, using our brains to analyse the scene, and coming to the conclusion about what objects are in that scene. 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:



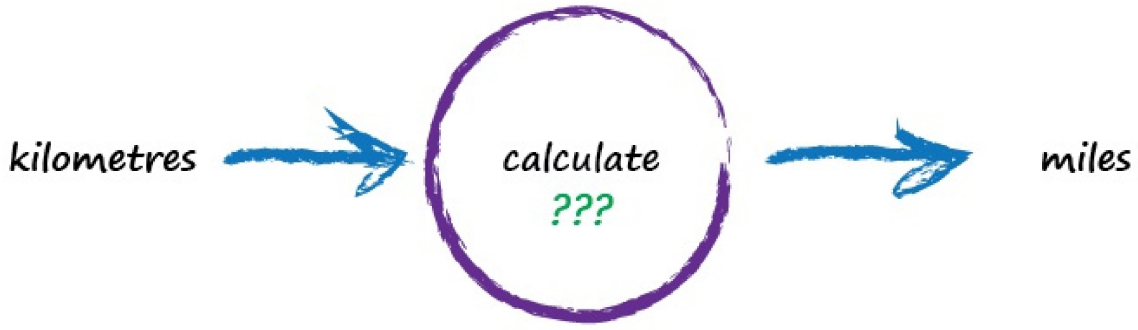
A computer takes some input, does some calculation and pops out an output. The following illustrates this. An input of "3 x 4" is processed, perhaps by turning multiplication into an easier set of additions, and the output answer "12" pops out.



"That's not so impressive!" you might be thinking. That's ok. We're using simple and familiar examples here to set out concepts which will apply to the more interesting neural networks we look at later.

Let's ramp up the complexity just a tiny notch.

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



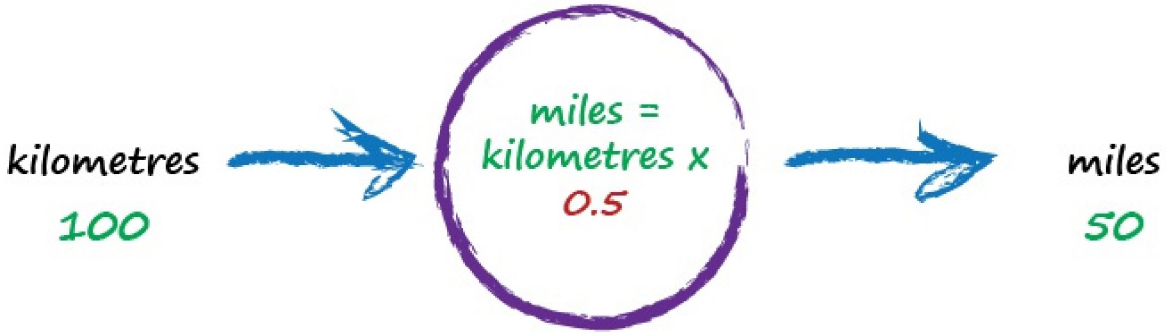
Now imagine we don't know the formula for converting between kilometres and miles. All we know is the the relationship between the two is **linear**. That means if we double the number in miles, the same distance in kilometres is also doubled. That makes intuitive sense. The universe would be a strange place if that wasn't true!

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.

The only other clues we have are some examples pairing kilometres with the correct value for miles. These are like real world observations used to test scientific theories - they're examples of real world truth.

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.

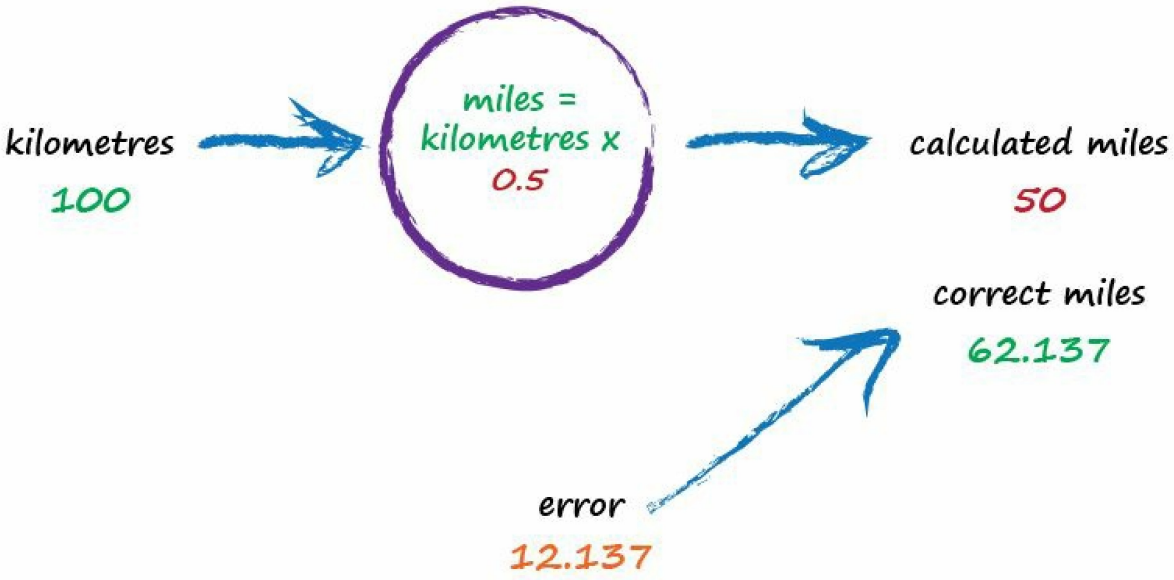


Here we have miles = kilometres x c, where kilometres is 100 and c is our current guess at 0.5. That gives 50 miles.

Okay. 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,

$$\begin{aligned} \textbf{error} &= \textbf{truth - calculated} \\ &= \textbf{62.137 - 50} \\ &= \textbf{12.137} \end{aligned}$$



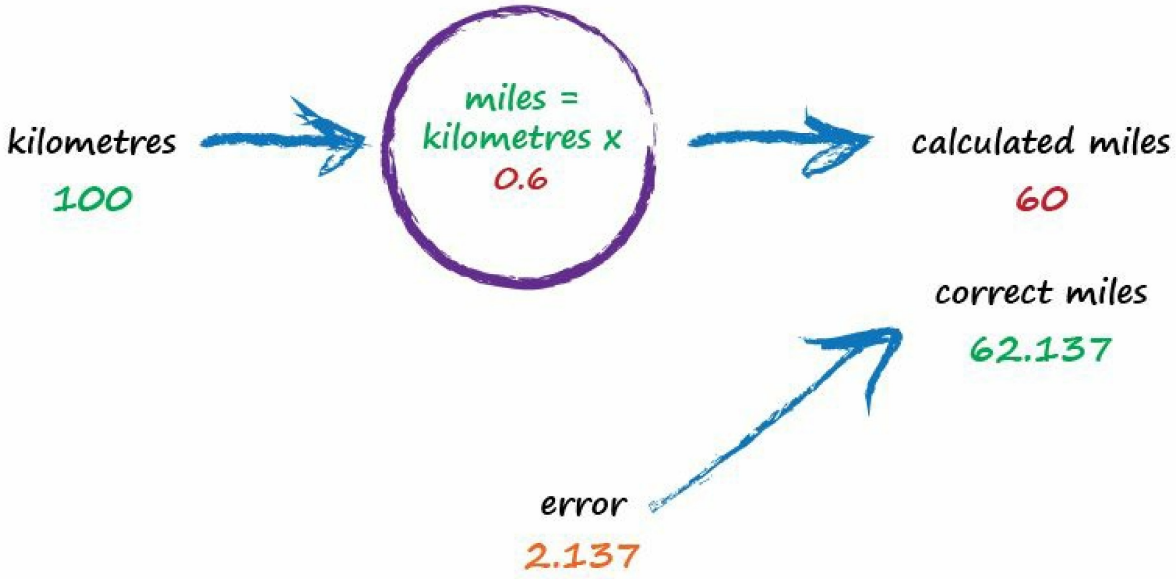
So what next? We know we're wrong, and by how much. Instead of being a reason to despair, we use this error to guide a second, better, guess at **c**.

Look at that error again. We were short by 12.137. Because the formula for converting kilometres to miles is linear, miles = kilometres x **c**, we know that increasing **c** will increase the output.

Let's nudge **c** up from 0.5 to 0.6 and see what happens.

With **c** now set to 0.6, we get miles = kilometres x **c** = 100 x 0.6 = 60. That's better than the previous answer of 50. We're clearly making progress!

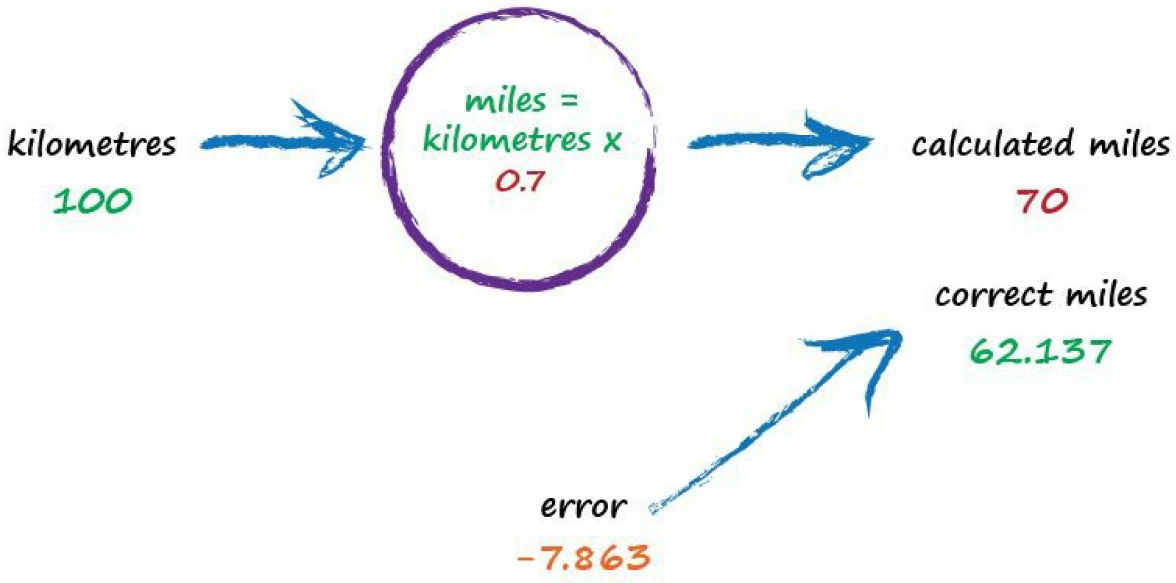
Now the error is a much smaller 2.137. It might even be an error we're happy to live with.



The important point here is that we used the error to guide how we nudged the value of c. We wanted to increase the output from 50 so we increased **c** a little bit.

Rather than try to use algebra to work out the exact amount **c** needs to change, let's continue with this approach of refining c. If you're not convinced, and think it's easy enough to work out the exact answer, remember that many more interesting problems won't have simple mathematical formulae relating the output and input. That's why we need more sophisticated methods - like neural networks.

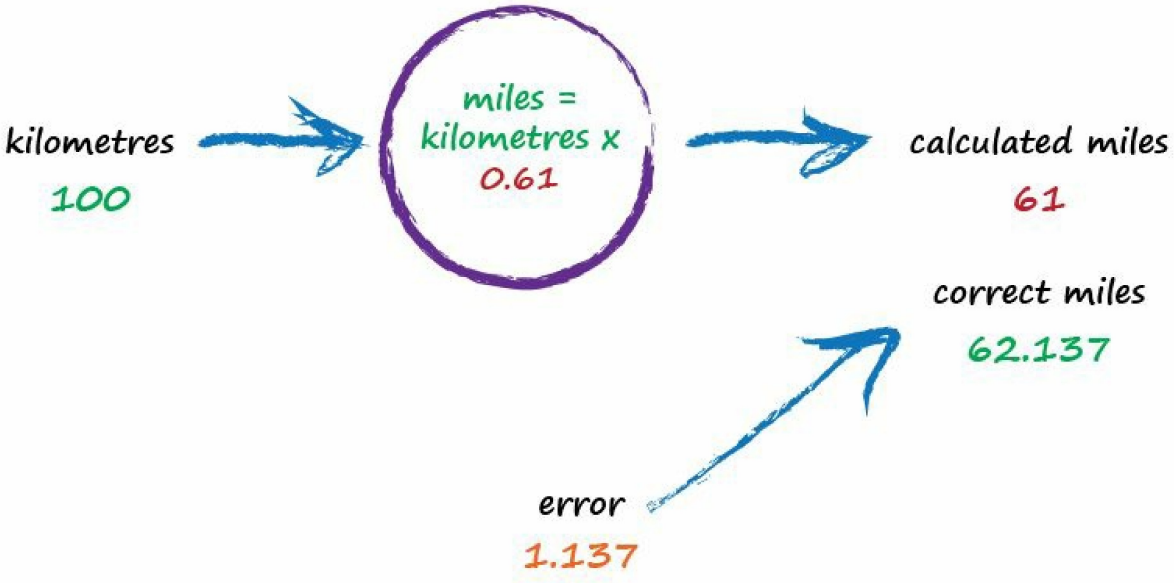
Let's do this again. The output of 60 is still too small. Let's nudge the value of **c** up again from 0.6 to 0.7.



Oh no! 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).

Ok so **c** = 0.6 was way better than c = 0.7. We could be happy with the small error from **c** = 0.6 and end this exercise now. But let's go on for just a bit longer. Why don't we nudge **c** up by just a tiny

amount, from 0.6 to 0.61.



That's much much better than before. We have an output value of 61 which is only wrong by 1.137 from the correct 62.137.

So that last effort taught us that we should moderate how much we nudge the value of c. If the outputs are getting close to the correct answer - that is, the error is getting smaller - then don't nudge the changeable bit so much. That way we avoid overshooting the right value, like we did earlier.

Again 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.

It is worth pausing to reflect on that - we've not solved a problem exactly in one step, like we often do in school maths or science problems. Instead, we've taken a very different approach by trying an answer and improving it repeatedly. Some use the term **iterative** and it means repeatedly improving an answer bit by bit.

Key Points:

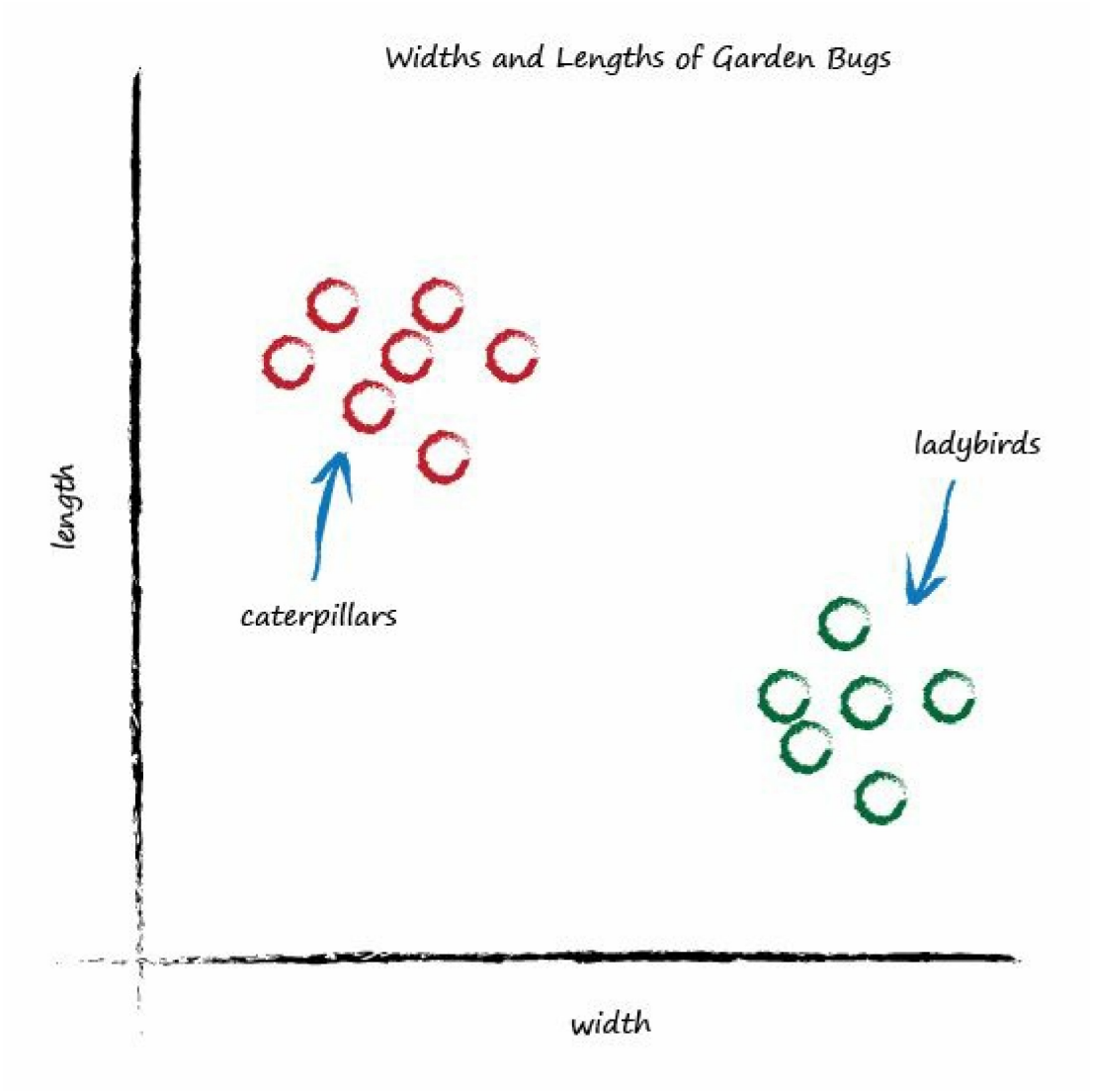
- All useful computer systems have an input, and an output, with some kind of calculation in between. Neural networks are no different.
- When we don't know exactly how something works we can try to estimate it with a model which includes parameters which we can adjust. If we didn't know how to convert kilometres to miles, we might use a linear function as a model, with an adjustable gradient.
- A good way of refining these models is to adjust the parameters based on how wrong the model is

compared to known true examples.

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.

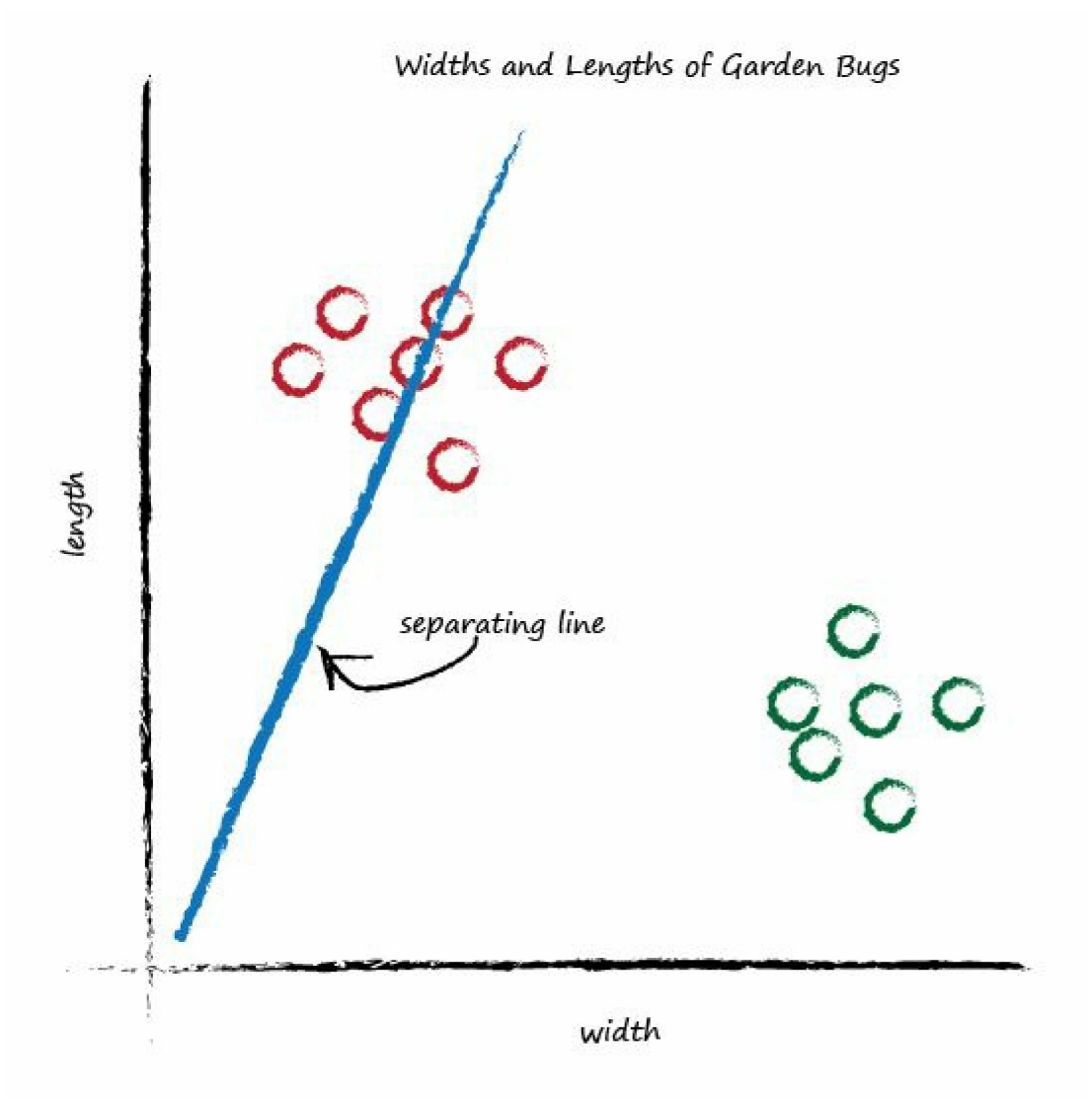
Now look at the following graph showing the measured widths and lengths of garden bugs.



You can clearly see two groups. The caterpillars are thin and long, and the ladybirds are wide and short.

Remember the predictor that tried to work out the correct number of miles given kilometres? That predictor had an adjustable linear function at it's heart. Remember, linear functions give straight lines when you plot their output against input. The adjustable parameter **c** changed the slope of that straight line.

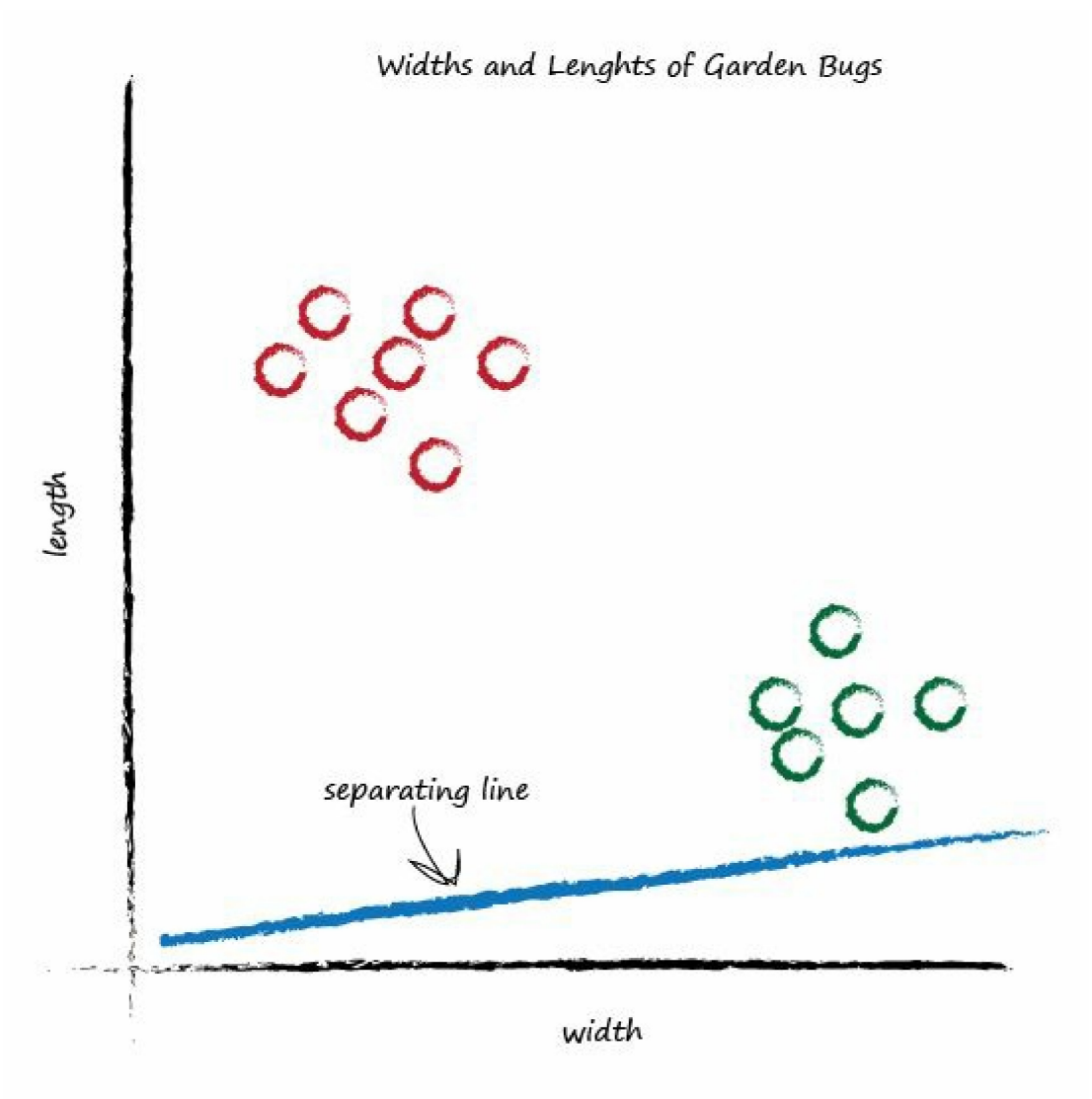
What happens if we place a straight line over that plot?



We can't use the line in the same way we did before - to convert one number (kilometres) into another (miles), but perhaps we can use the line to separate different kinds of things.

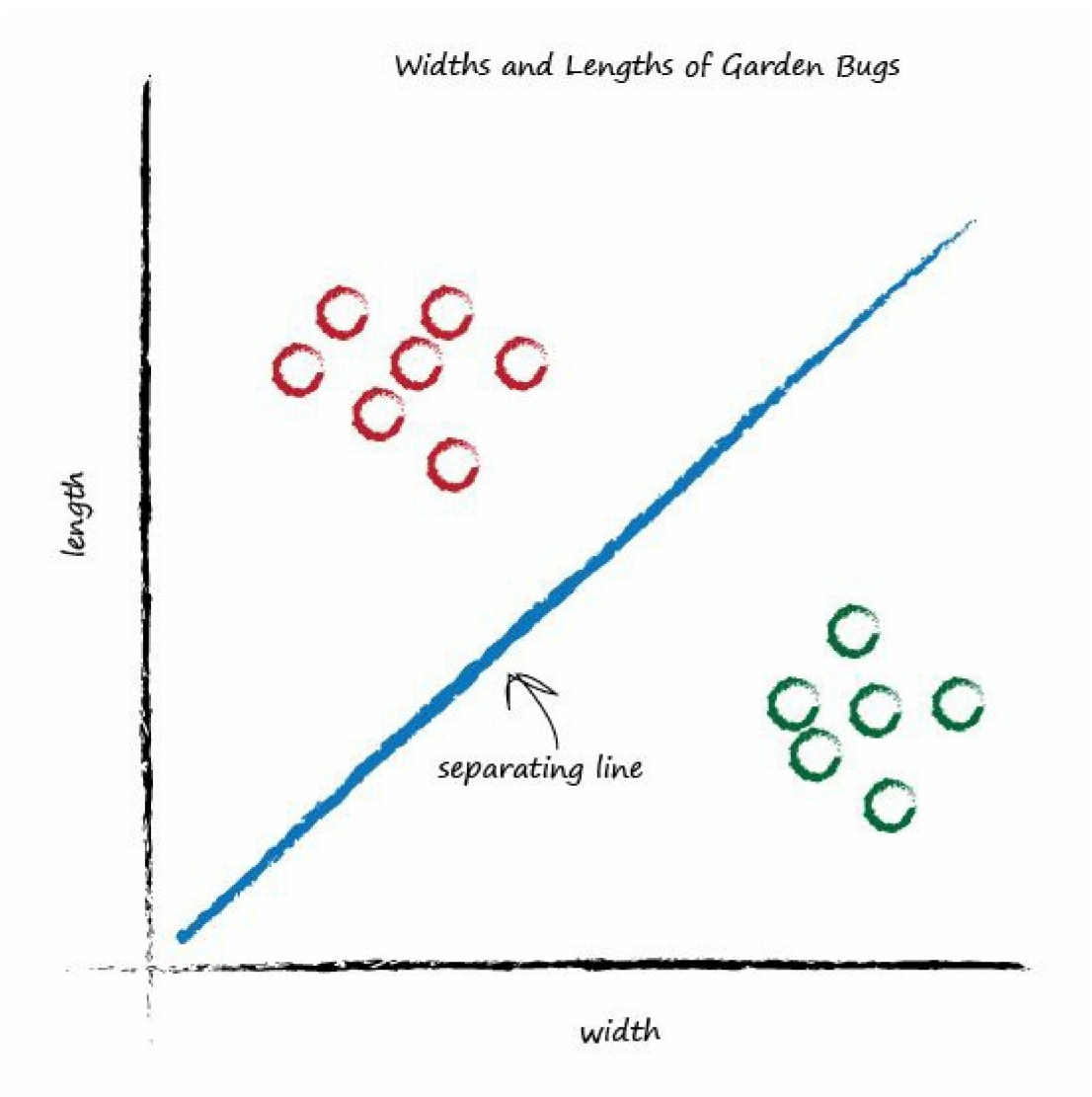
In the plot above, if the line was dividing the caterpillars from the ladybirds, then it could be used to **classify** an unknown bug based on its measurements. The line above doesn't do this yet because half the caterpillars are on the same side of the dividing line as the ladybirds.

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



This time the line is even less useful! It doesn't separate the two kinds of bugs at all.

Let's have another go:

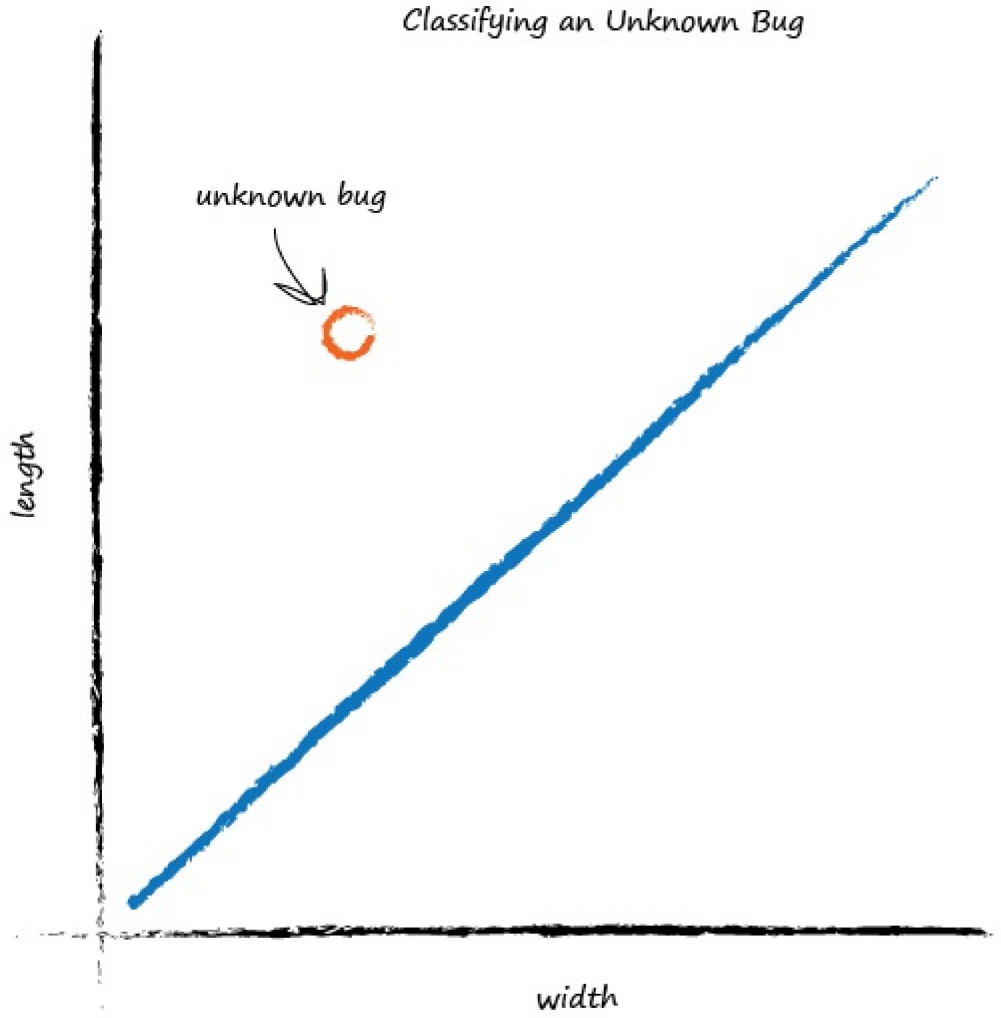

Widths and Lengths of Garden Bugs
length
separating line
width

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.

Look at the following plot, you can see the unknown bug is a caterpillar because it lies above the line. This classification is simple but pretty powerful already!



We've seen how a linear function inside our simple predictors can be used to classify previously unseen data.

But we've skipped over a crucial element. How do we get the right slope? How do we improve a line we know isn't a good divider between the two kinds of bugs?

The answer to that is again at the very heart of how neural networks learn, and we'll look at this next.

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.

How do we do this?

Rather than develop some mathematical theory upfront, let's try to feel our way forward by trying to do it. We'll understand the mathematics better that way.

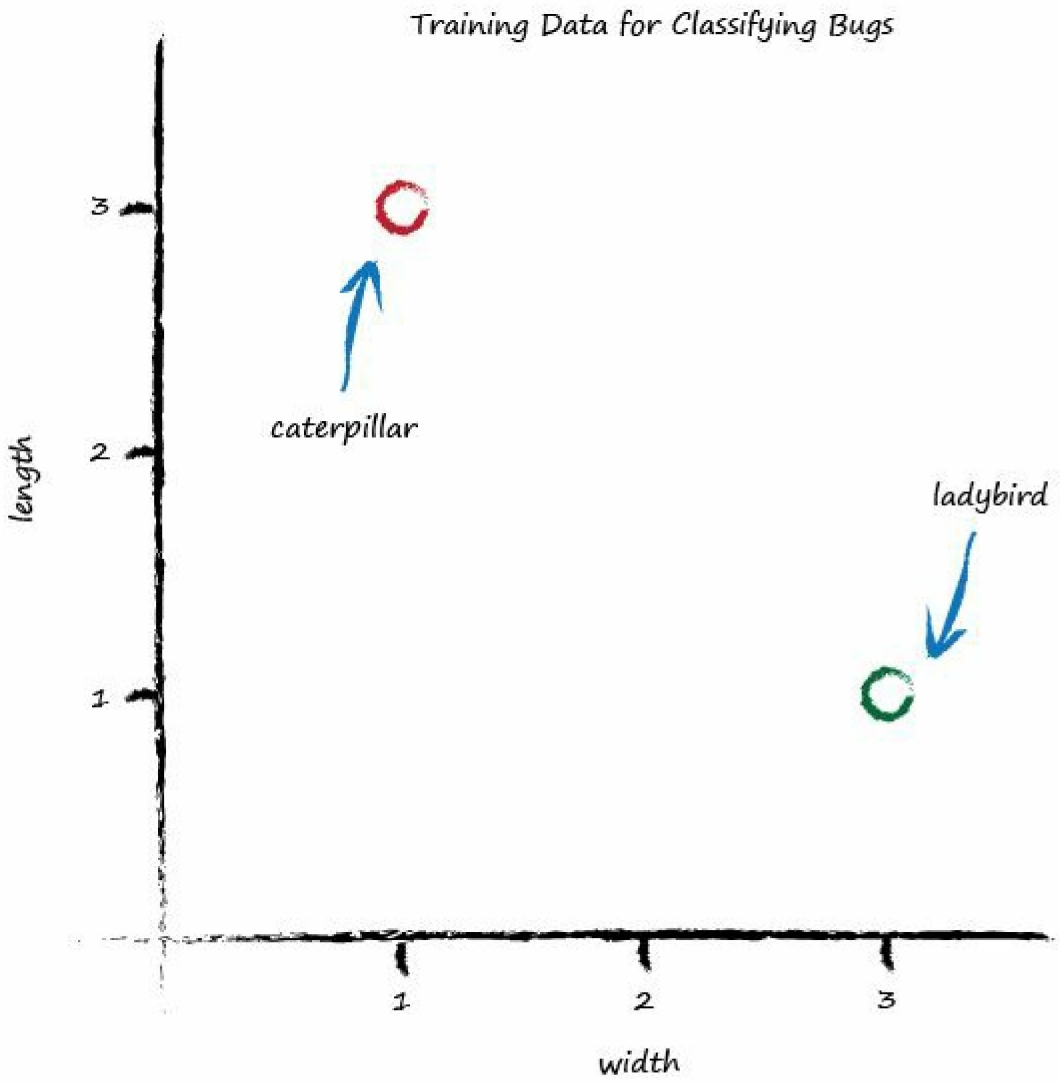
We do need some examples to learn from. The following table shows two examples, just to keep this exercise simple.

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

We have an example of a bug which has width 3.0 and length 1.0, which we know is a ladybird. We also have an example of a bug which is longer at 3.0 and thinner at 1.0, which is a caterpillar.

This is a set of examples which we know to be the truth. It is these examples which will help refine the slope of the classifier function. Examples of truth used to teach a predictor or a classifier are called the **training data**.

Let's plot these two training data examples. Visualising data is often very helpful to get a better understand of it, a feel for it, which isn't easy to get just by looking at a list or table of numbers.



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:

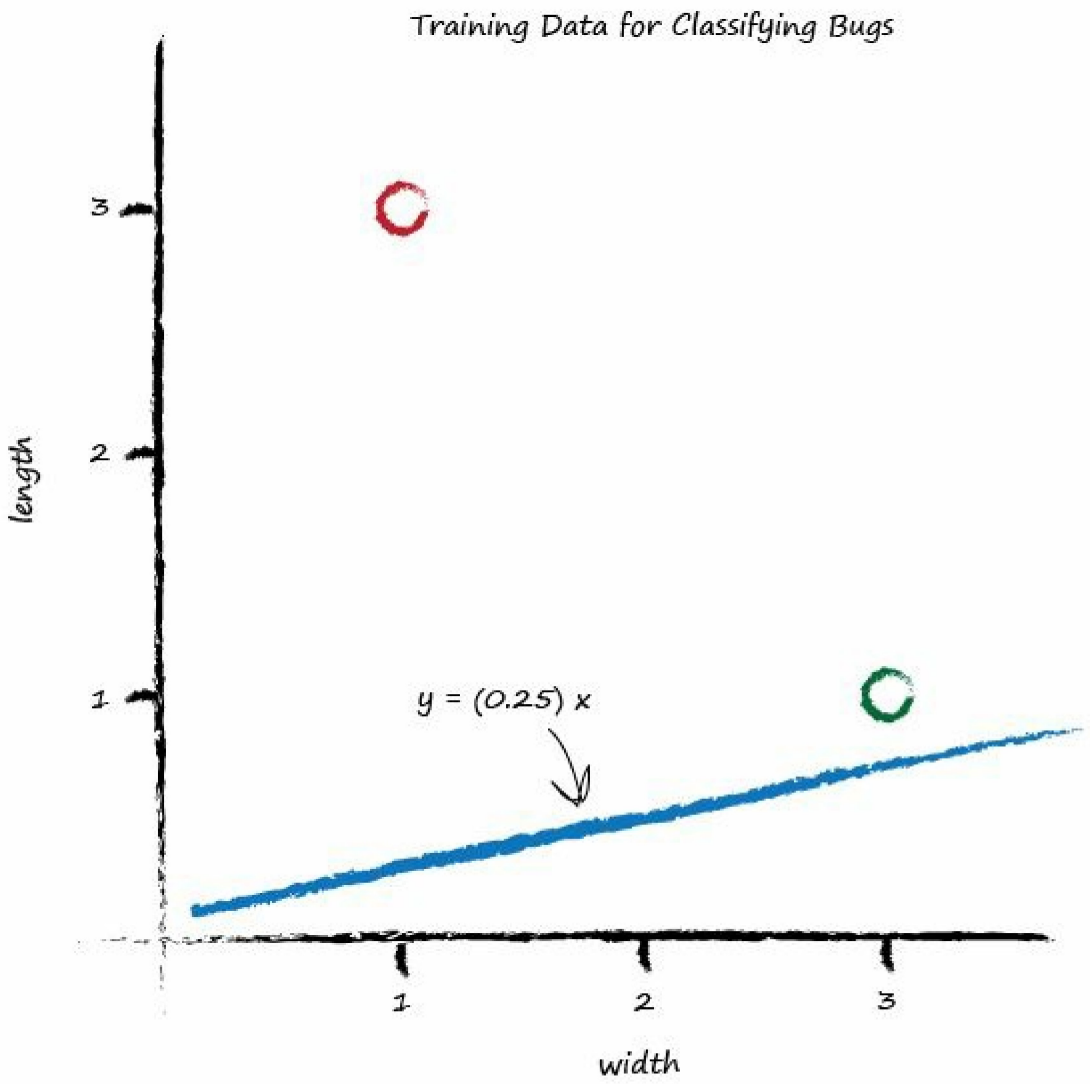
$$y = Ax$$

We've deliberately used the names **y** and **x** instead of length and width, because strictly speaking, the line is not a predictor here. It doesn't convert width to length, like we previously converted miles to kilometres. Instead, it is a dividing line, a classifier.

You may also notice that this $y = Ax$ is simpler than the fuller form for a straight line $y = Ax + B$. We've deliberately kept this garden bug scenario as simple as possible. Having a non-zero **B** simple means the line doesn't go through the origin of the graph, which doesn't add anything useful to our scenario.

We saw before 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:



Well, we can see that the line **y = 0.25x** isn't a good classifier already without the need to do any calculations. The line doesn't divide the two types of bug. We can't say "if the bug is above the line then it is a caterpillar" because the ladybird is above the line too.

So intuitively we need to move the line up a bit. We'll resist the temptation to do this by looking at the plot and drawing a suitable line. We want to see if we can find a repeatable recipe to do this, a series of computer instructions, which computer scientists call an **algorithm**.

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

$$\mathbf{y = (0.25) * (3.0) = 0.75}$$

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 line 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. It's just a small number above 1.0, We could have chosen 1.2, or even 1.3, but we don't want a larger number like 10 or 100 because that would make it more likely that the line goes above both ladybirds and caterpillars, resulting in a separator that wasn't useful at all.

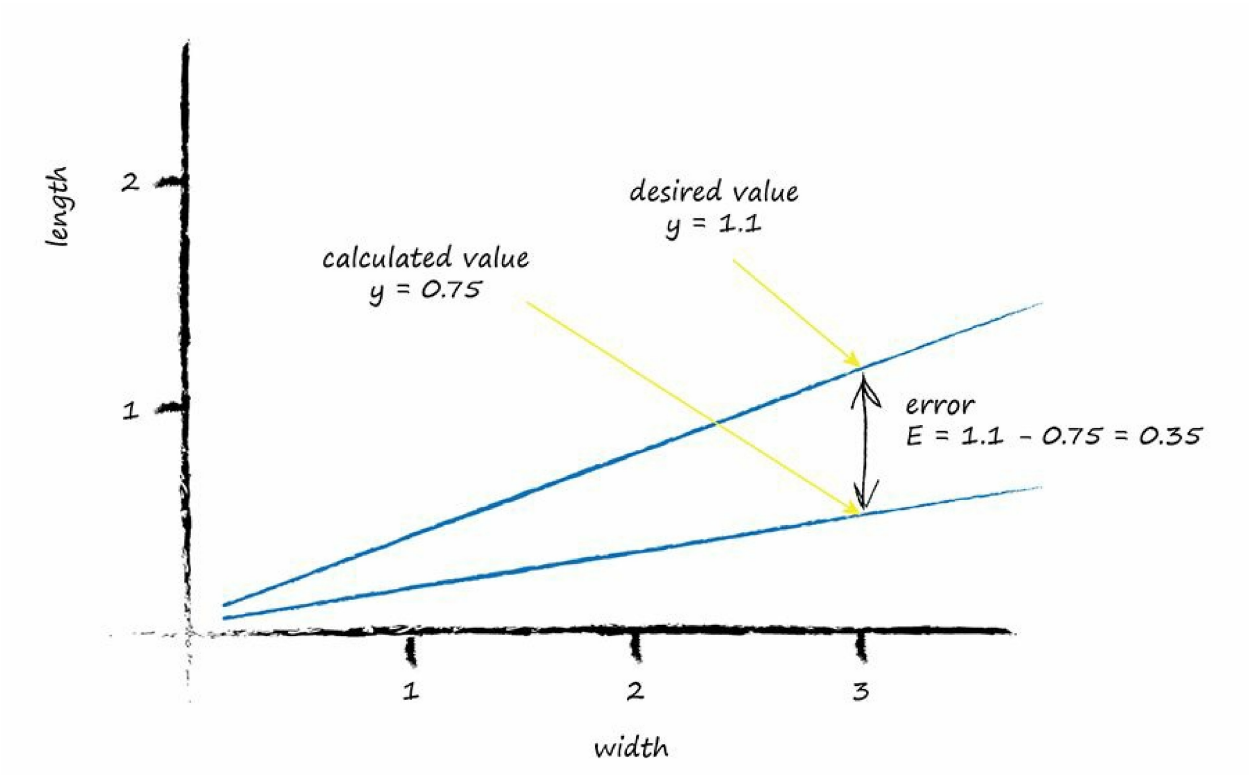
So the desired target is 1.1, and the error **E** is

error = (desired target - actual output)

Which is,

E = 1.1 - 0.75 = 0.35

Let's pause and have a remind ourselves what the error, the desired target 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**? If we can know this, then we can understand how changing one affects the other.

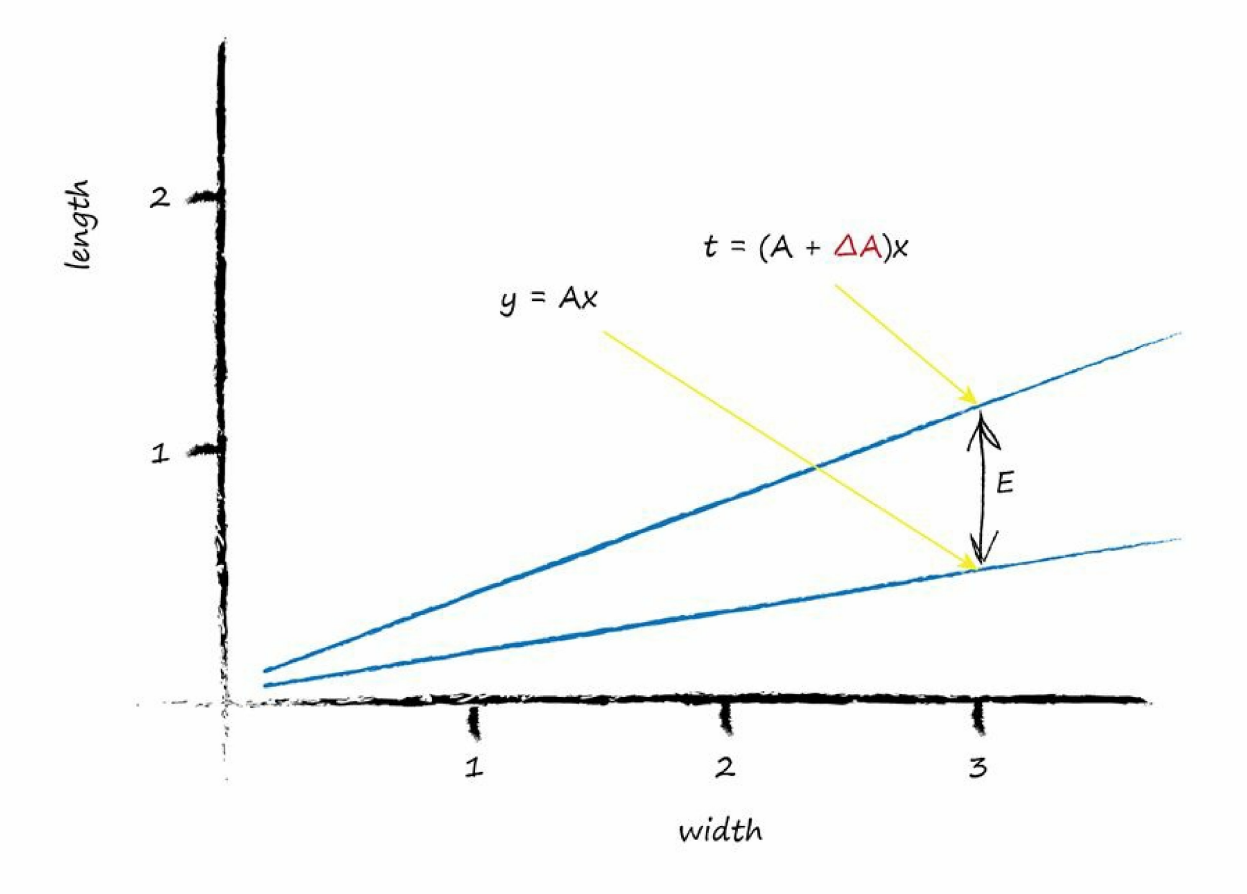
Let's start with the linear function for the classifier:

$$\mathbf{y = Ax}$$

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 delta symbol Δ to mean "a small change in". Let's write that out:

$$\mathbf{t = (A + \Delta A)x}$$

Let's picture this to make it easier to understand. You can see the new slope (**A** + Δ**A**).



Remember the error **E** was the difference between the desired correct value and the one we calculate based on our current guess for **A**. That is, **E** was **t - y**.

Let's write that out to make it clear:

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

Expanding out the terms and simplifying:

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

$$\mathbf{E = (\Delta A)x}$$

That's remarkable! The error **E** is related to Δ**A** is a very simple way. It's so simple that I thought it must be wrong - but it was indeed correct. Anyway, this simple relationship makes our job much

easier.

It's easy to get lost or distracted by that algebra. Let's remind ourselves of what we wanted to get out of all this, in plain English.

We wanted to know how much to adjust **A** by to improve the slope of the line so it is a better classifier, being informed by the error **E**. To do this we simply re-arrange that last equation to put Δ**A** on it's own:

$$\Delta \mathbf{A} = \mathbf{E} / \mathbf{x}$$

That's it! That's the magic expression we've been looking for. We can use the error **E** to refine the slope **A** of the classifying line by an amount Δ**A**.

Let's do it - let's update that initial slope.

The error was 0.35 and the **x** was 3.0. That gives Δ**A** = **E** / **x** as 0.35 / 3.0 = 0.1167. That means we need to change the current **A** = 0.25 by 0.1167. That means the new improved value for **A** is (**A** + Δ**A**) which is 0.25 + 0.1167 = 0.3667. As it happens, the calculated value of **y** with this new **A** is 1.1 as you'd expect - it's the desired target value.

Phew! We did it! All that work, and we have a method for refining that parameter **A**, informed by the current error.

Let's press on.

Now we're done with one training example, let's learn from the next one. Here we have a known true pairing of **x** = 1.0 and **y** = 3.0.

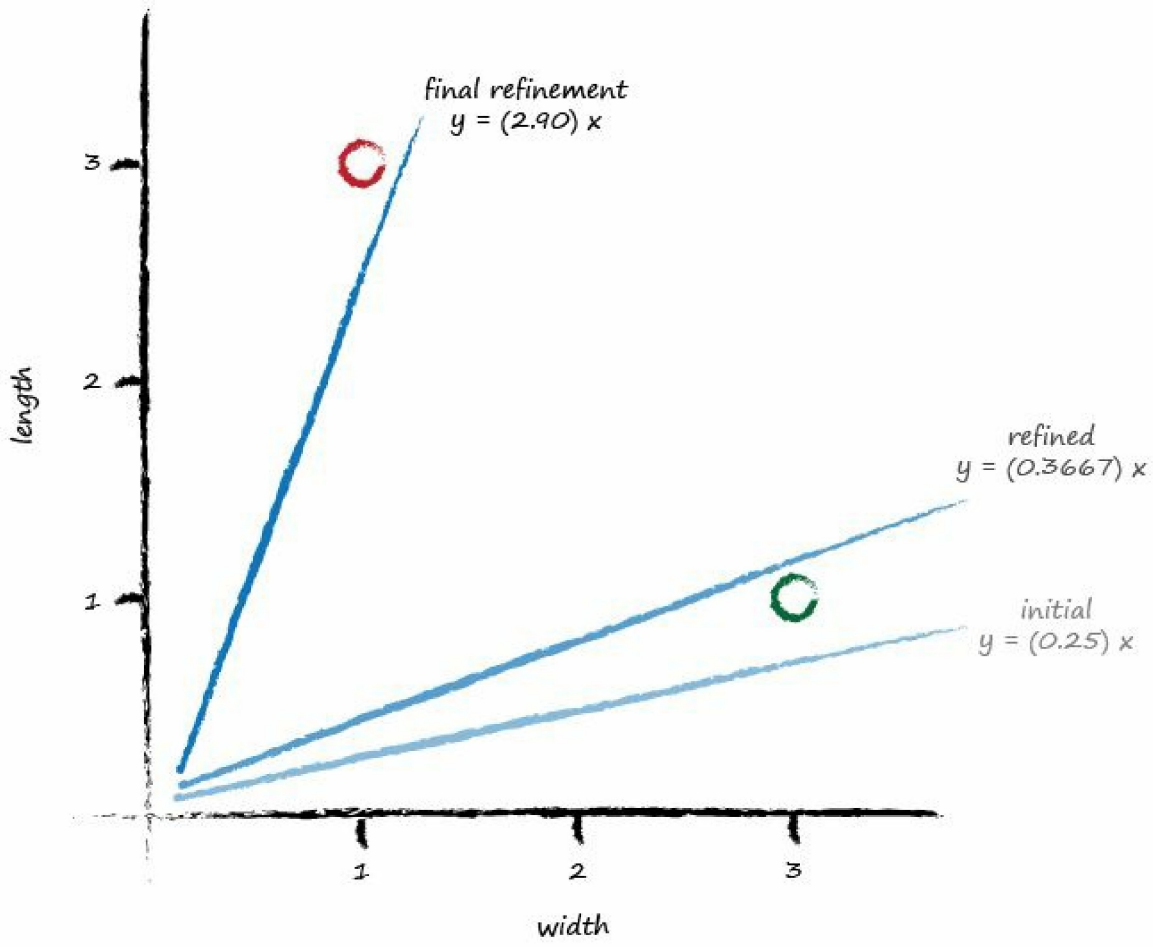
Let's see what happens when we put **x** = 1.0 into the linear function which is now using the updated **A** = 0.3667. We get **y** = 0.3667 * 1.0 = 0.3667. That's not very close to the training example with **y** = 3.0 at all.

Using the same reasoning as before that we want the line to not cross the training data but instead be just above or below it, 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.

That's a bigger error than before, but if you think about it, all we've had so far for the linear function to learn from is a single training example, which clearly biases the line towards that single example.

Let's update the **A** again, just like we did before. The Δ**A** is **E** / **x** which is 2.5333 / 1.0 = 2.5333. That means the even newer **A** is 0.3667 + 2.5333 = 2.9. That means for **x** = 1.0 the function gives 2.9 as the answer, which is what the desired value was.

That's a fair amount of working out so let's pause again and visualise what we've done. The following plot shows the initial line, the line updated after learning from the first training example, and the final line after learning from the second training example.



Wait! What's happened! Looking at that plot, we don't seem to have improved the slope in the way we had hoped. It hasn't divided neatly the region between ladybirds and caterpillars.

Well, we got what we asked for. The line updates to give each desired value for y.

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. That is, we calm them down a bit. Instead of jumping enthusiastically to each new **A**, we take a fraction of the change $\Delta\mathbf{A}$, not all of it. This way we move in the direction that the training example suggests, but do so slightly cautiously, keeping some of the previous value which was arrived at through potentially many previous training iterations. We saw this idea of moderating our refinements before - with the simpler miles to kilometres predictor, where we nudged the parameter **c** as a fraction of the actual error.

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, both of which are normal in real world measurements, 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:

$$\Delta \mathbf{A} = \mathbf{L}\,(\mathbf{E} / \mathbf{x})$$

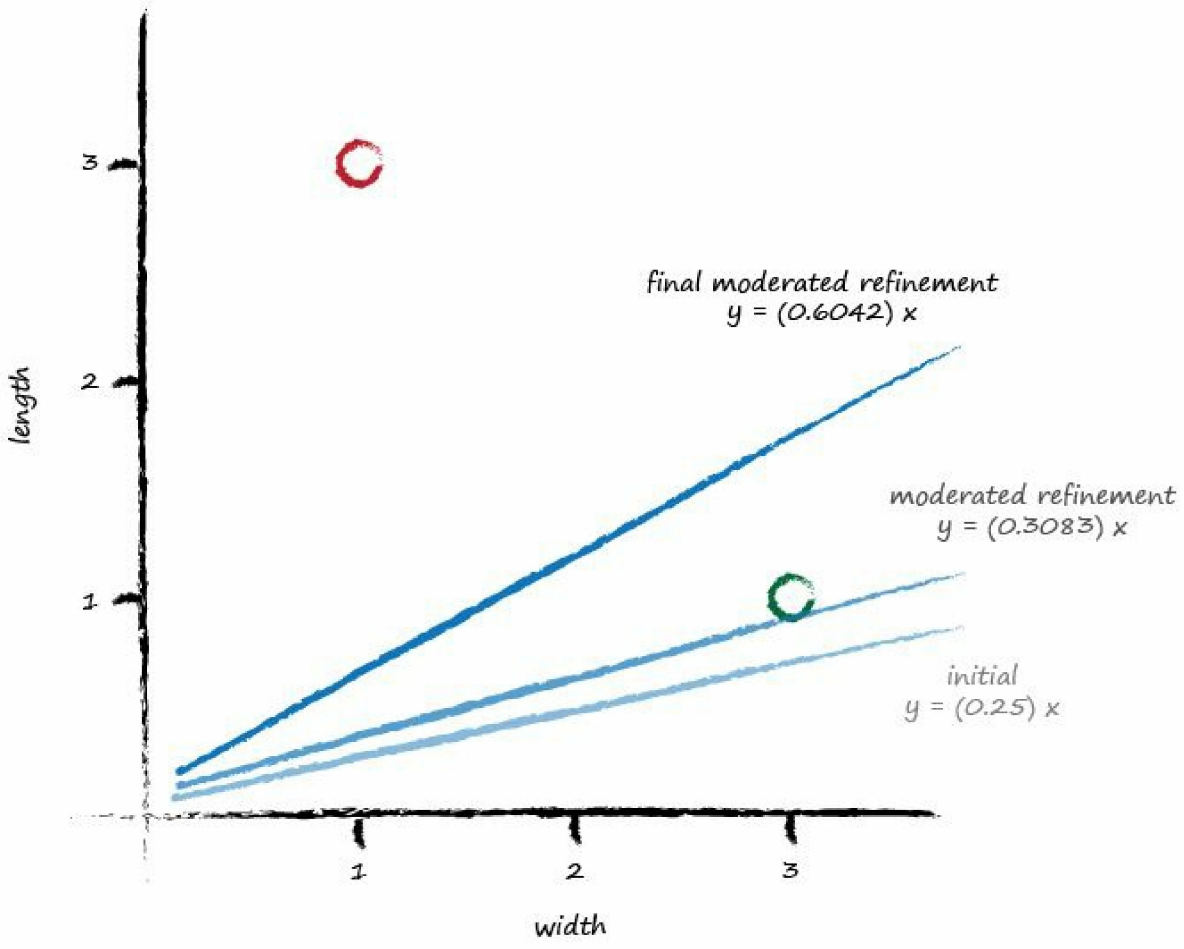
The moderating factor is often called a **learning rate**, and we've called it **L**. Let's pick **L** = 0.5 as a reasonable fraction just to get started. It simply means we only update half as much as would have done without moderation.

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 $\Delta\mathbf{A} = \mathbf{L}\,(\mathbf{E} / \mathbf{x}) = 0.5 * 0.35 / 3.0 = 0.0583$. 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 but it's not a bad result if you consider it a first refinement step of many to come. It did move in the right direction away from the initial line.

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 0.9 so the error is (0.9 = 0.3083) = 0.5917. The $\Delta\mathbf{A} = \mathbf{L}\,(\mathbf{E} / \mathbf{x}) = 0.5 * 0.5917 / 1.0 = 0.2958$. The even newer **A** is now 0.3083 + 0.2958 = 0.6042.

Let's visualise again the initial, improved and final line to see if moderating updates leads to a better dividing line between ladybird and caterpillar regions.



This is really good!

Even with these two simple training examples, and a relatively simple update method using a moderating **learning rate**, we have very rapidly arrived at a good dividing line **y** = **Ax** where **A** is 0.6042.

Let's not diminish what we've achieved. We've achieved an automated method of learning to classify from examples that is remarkably effective given the simplicity of the approach.

Brilliant!

Key Points:

- We can use simple maths to understand the relationship between the output error of a linear classifier and the adjustable slope parameter. That is the same as knowing how much to adjust the slope to remove that output error.
- A problem with doing these adjustments naively, is that the model is updated to best match the last training example only, discarding all previous training examples. A good way to fix this is to moderate the updates with a learning rate so no single training example totally dominates the learning.
- Training examples from the real world can be noisy or contain errors. Moderating updates in this

way helpfully limits the impact of these false examples.

Sometimes One Classifier Is Not Enough

The simple predictors and classifiers we've worked with so far - the ones that takes some input, do some calculation, and throw out an answer - although pretty effective as we've just seen, are not enough to solve some of the more interesting problems we hope to apply neural networks to.

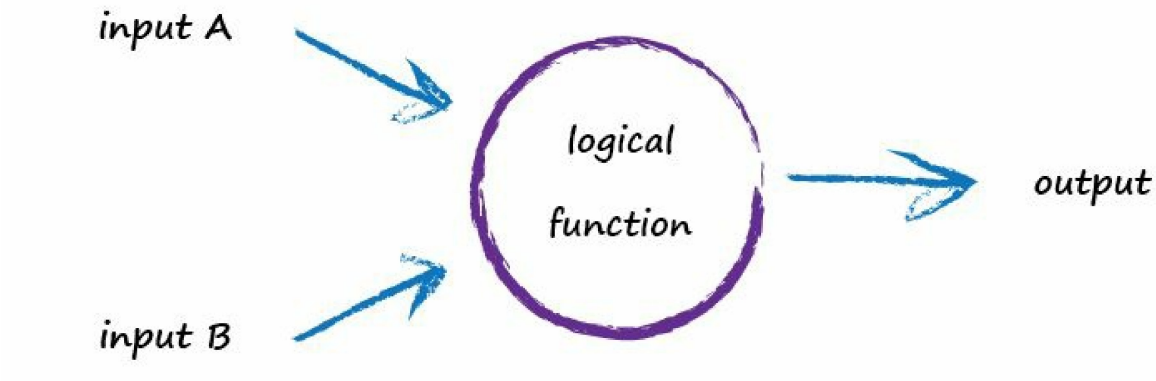
Here we'll illustrate the limit of a linear classifier with a simple but stark example. Why do we want to do this, and not jump straight to discussing neural networks? The reason is that a key design feature of neural networks comes from understanding this limit - so worth spending a little time on.

We'll be moving away from garden bugs and looking at **Boolean** logic functions. If that sounds like mumbo jumbo jargon - don't worry. George Boole was a mathematician and philosopher, and his name is associated with simple functions like AND and OR.

Boolean logic functions are like language or thought functions. If we say "you can have your pudding only if you've eaten your vegetables AND if you're still hungry" we're using the Boolean AND function. The Boolean AND is only true if both conditions are true. It's not true if only one of them is true. So if I'm hungry, but haven't eaten my vegetables, then I can't have my pudding.

Similarly, if we say "you can play in the park if it's the weekend OR you're on annual leave from work" we're using the Boolean OR function. The Boolean OR is true if any, or all, of the conditions are true. They don't all have to be true like the Boolean AND function. So if it's not the weekend, but I have booked annual leave, I can indeed go and play in the park.

If we think back to our first look at functions, we saw them as a machine that took some inputs, did some work, and output an answer. Boolean logical functions typically take two inputs and output one answer:



Computers often represent **true** as the number 1, and **false** as the number 0. The following table shows the logical AND and OR functions using this more concise notation for all combinations of inputs A and B.

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

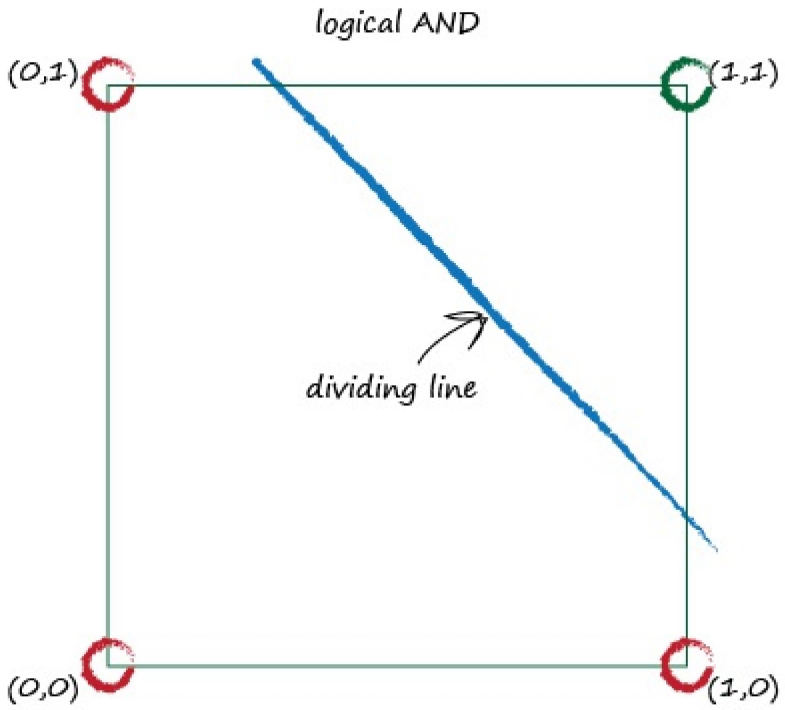
1	1	1	1

You can see quite clearly, that the AND function is only true if both A and B are true. Similarly you can see that OR is true whenever any of the inputs A or B is true.

Boolean logic functions are really important in computer science, and in fact the earliest electronic computers were built from tiny electrical circuits that performed these logical functions. Even arithmetic was done using combinations of circuits which themselves were simple Boolean logic functions.

Imagine using a simple linear classifier to learn from training data whether the data was governed by a Boolean logic function. That's a natural and useful thing to do for scientists wanting to find causal links or correlations between some observations and others. For example, is there more malaria when it rains AND it is hotter than 35 degrees? Is there more malaria when either (Boolean OR) of these conditions is true?

Look at the following plot, showing the two inputs A and B to the logical function as coordinates on a graph. The plot shows that only when both are true, with value 1, is the output also true, shown as green. False outputs are shown red.



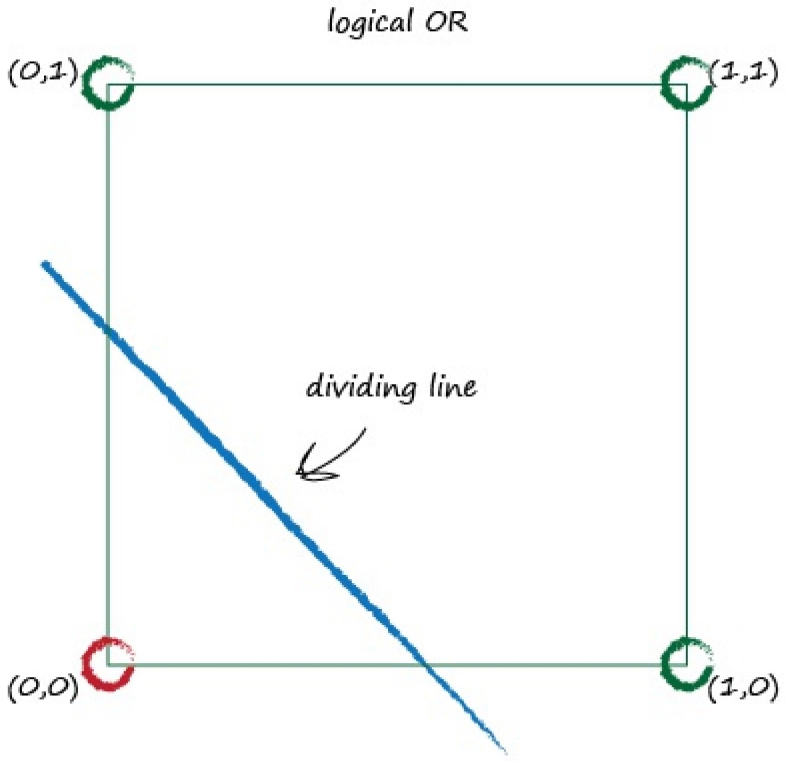
You can also see a straight line that divides the red from the green regions. That line is a linear function that a linear classifier could learn, just as we have done earlier.

We won't go through the numerical workings out as we did before because they're not fundamentally different in this example.

In fact there are many variations on this dividing line that would work just as well, but the main point is that it is indeed possible for a simple linear classifier of the form **y** = **ax**+**b** to learn the Boolean

AND function.

Now look at the Boolean OR function plotted in a similar way:

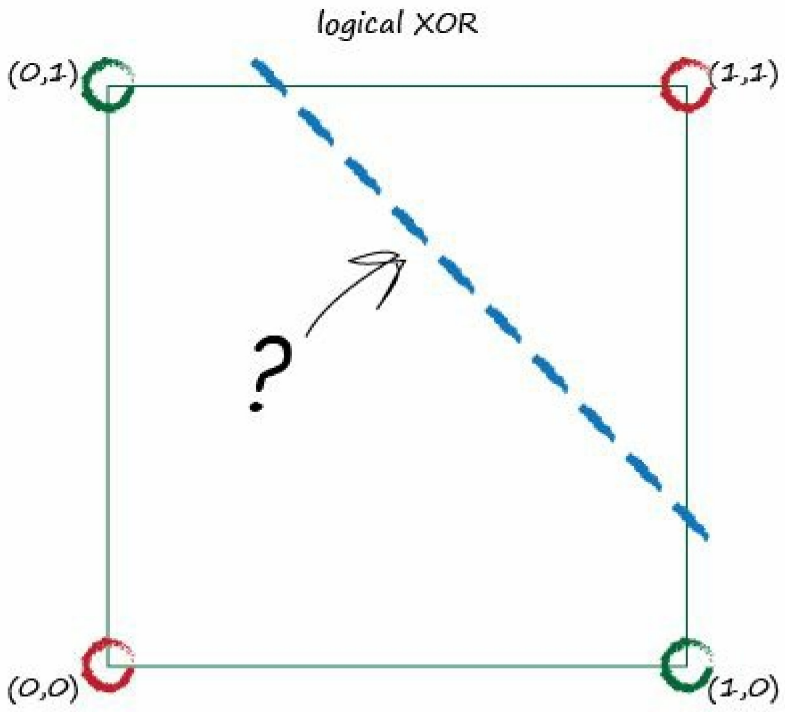


This time only the (0,0) point is red because it corresponds to both inputs A and B being false. All other combinations have at least one A or B as true, and so the output is true. The beauty of the diagram is that it makes clear that it is possible for a linear classifier to learn the Boolean OR function, too.

There is another Boolean function called XOR, short for eXclusive OR, which only has a true output if either one of the inputs A or B is true, but not both. That is, when the inputs are both false, or both true, the output is false. The following table summarises this:

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

Now look at a plot of the of this function on a grid with the outputs coloured:



This is a challenge! We can't seem to separate the red from the blue regions with only a single straight dividing line.

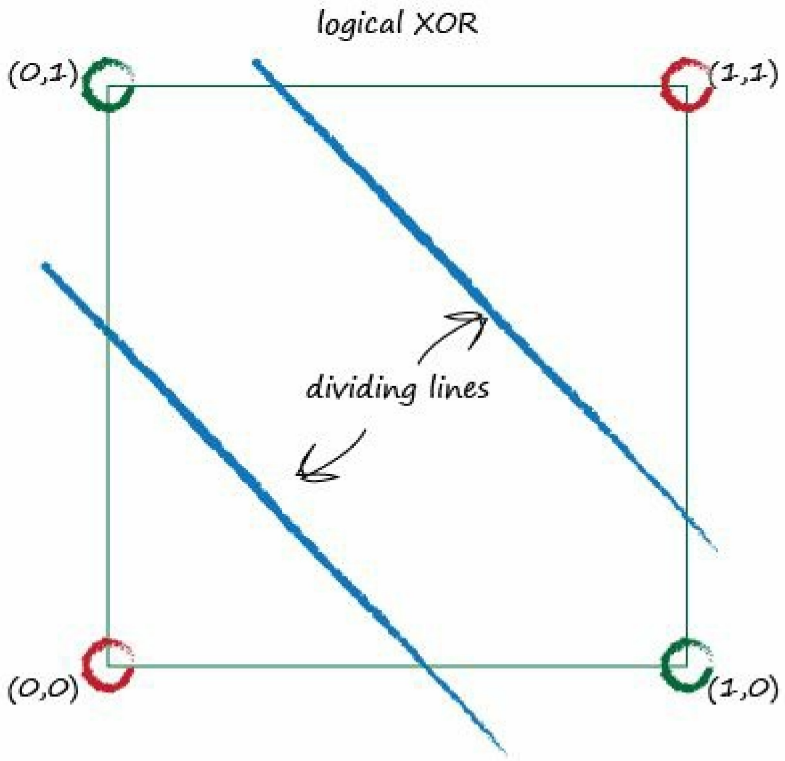
It is, in fact, impossible to have a single straight line that successfully divides the red from the green regions for the Boolean XOR. That is, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function.

We've just illustrated a major limitation of the simple linear classifier. A simple linear classifier is not useful if the underlying problem is not separable by a straight line.

We want neural networks to be useful for the many many tasks where the underlying problem is not linearly separable - where a single straight line doesn't help.

So we need a fix.

Luckily the fix is easy. In fact the diagram below which has two straight lines to separate out the different regions suggests the fix - we use multiple classifiers working together. That's an idea central to neural networks. You can imagine already that many linear lines can start to separate off even unusually shaped regions for classification.



Before we dive into building neural networks made of many classifiers working together, let's go back to nature and look at the animal brains that inspired the neural network approach.

Key Points:

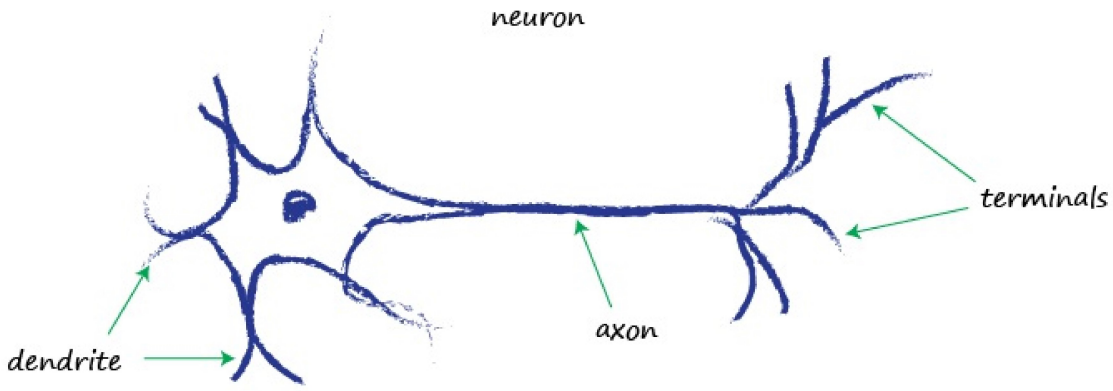
- A simple linear classifier can't separate data where that data itself isn't governed by a single linear process. For example, data governed by the logical XOR operator illustrates this.
- However the solution is easy, you just use multiple linear classifiers to divide up data that can't be separated by a single straight dividing line.

Neurons, Nature's Computing Machines

We said earlier that animal brains puzzled scientists, because even small ones like a pigeon brain were vastly more capable than digital computers with huge numbers of electronic computing elements, huge storage space, and all running at frequencies much faster than fleshy squishy natural brains.

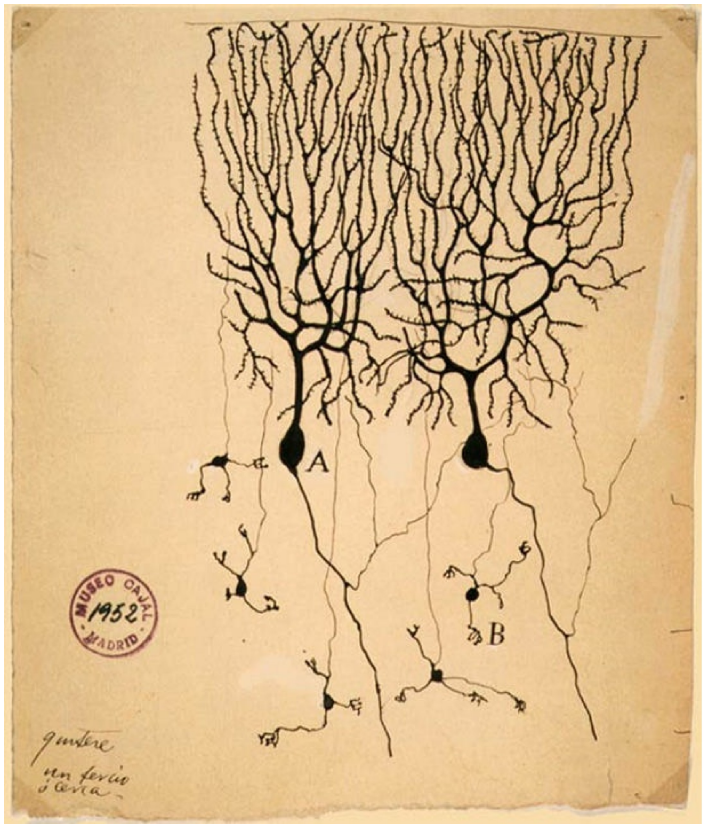
Attention turned to the architectural differences. Traditional computers processed data very much sequentially, and in pretty exact concrete terms. There is no fuzziness or ambiguity about their cold hard calculations. Animal brains, on the other hand, although apparently running at much slower rhythms, seemed to process signals in parallel, and fuzziness was a feature of their computation.

Let's look at the basic unit of a biological brain - the **neuron**:



Neurons, although there are various forms of them, all transmit an electrical signal from one end to the other, from the dendrites along the axons to the terminals. These signals are then passed from one neuron to another. This is how your body senses light, sound, touch pressure, heat and so on. Signals from specialised sensory neurons are transmitted along your nervous system to your brain, which itself is mostly made of neurons too.

The following is a sketch of neurons in a pigeon's brain, made by a Spanish neuroscientist in 1889. You can see the key parts - the dendrites and the terminals.



How many neurons do we need to perform interesting, more complex, tasks?

Well, the very capable human brain has about 100 billion neurons! A fruit fly has about 100,000 neurons and is capable of flying, feeding, evading danger, finding food, and many more fairly complex tasks. This number, 100,000 neurons, is well within the realm of modern computers to try to replicate. A nematode worm has just 302 neurons, which is positively miniscule compared to today's digital computer resources! But that worm is able to do some fairly useful tasks that traditional computer programs of much larger size would struggle to do.

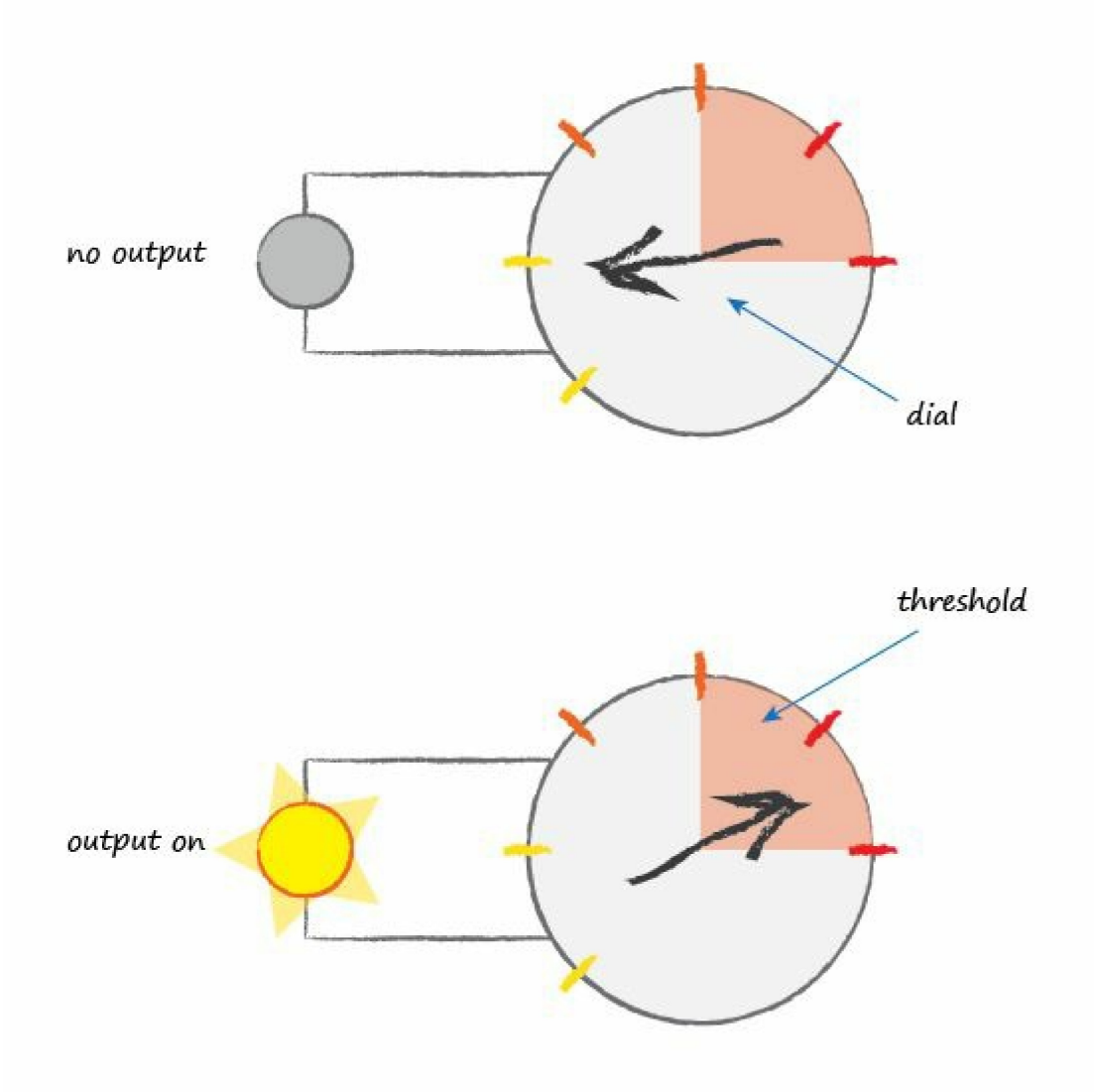
So what's the secret? Why are biological brains so capable given that they are much slower and consist of relatively few computing elements when compared to modern computers? The complete functioning of brains, consciousness for example, is still a mystery, but enough is known about neurons to suggest different ways of doing computation, that is, different ways to solve problems.

So let's look at how a neuron works. 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.

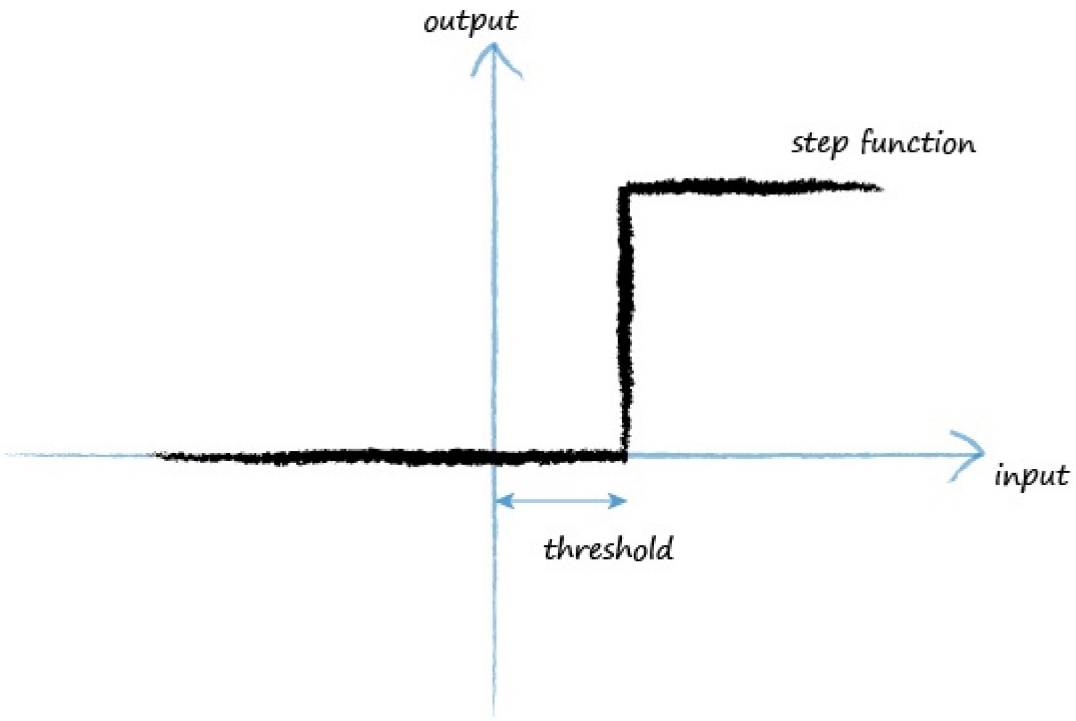
So could we represent neurons as linear functions, just like we did before? Good idea, but no. A biological neuron doesn't produce an output that is simply a simple linear function of the input. That is, its output does not take the form output = (constant * input) + (maybe another constant).

Observations suggest that neurons don't react readily, but instead suppress the input until it has grown so large that it triggers an output. You can think of this as a threshold that must be reached before any output is produced. It's like water in a cup - the water doesn't spill over until it has first filled the cup. Intuitively this makes sense - the neurons don't want to be passing on tiny noise signals, only

emphatically strong intentional signals. The following illustrates this idea of only producing an output signal if the input is sufficiently dialed up to pass a **threshold**.

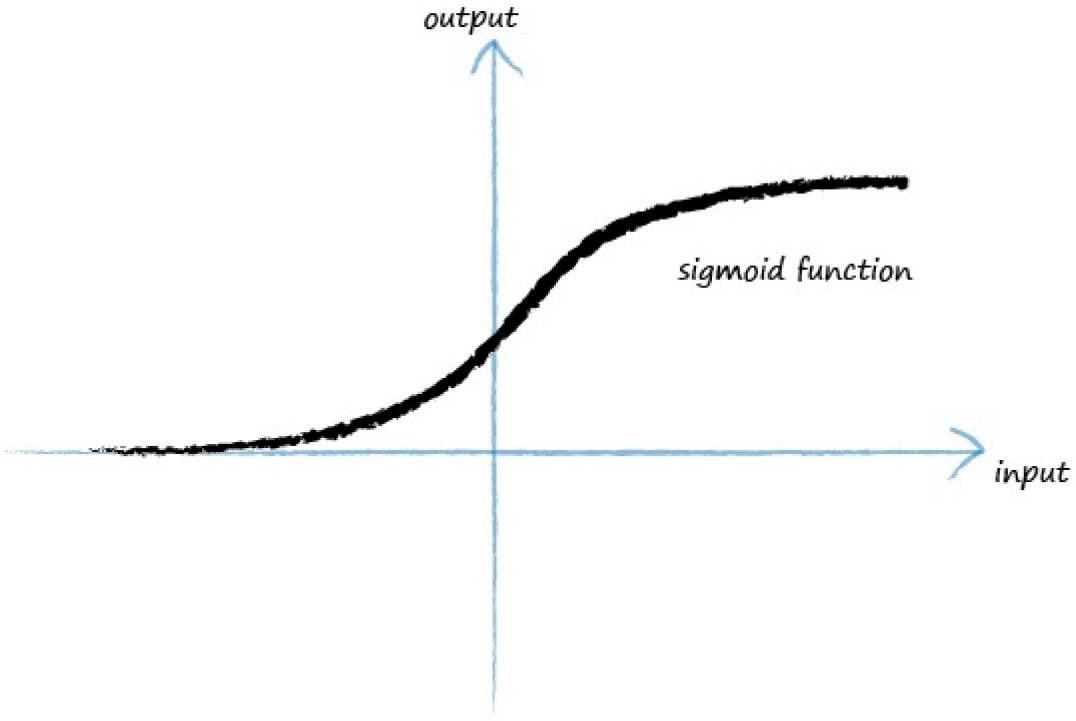


A function that takes the input signal and generates an output signal, but takes into account some kind of threshold is called an **activation function**. Mathematically, there are many such activation functions that could achieve this effect. A simple **step function** could do this:



You can see for low input values, the output is zero. However once the threshold input is reached, output jumps up. An artificial neuron behaving like this would be like a real biological neuron. The term used by scientists actually describes this well, they say that neurons **fire** when the input reaches the threshold.

We can improve on the step function. The S-shaped function shown below is called the **sigmoid function**. It is smoother than the cold hard step function, and this makes it more natural and realistic. Nature rarely has cold hard edges!



This smooth S-shaped sigmoid function is what we'll be continue to use for making our own neural network. Artificial intelligence researchers will also use other, similar looking functions, but the sigmoid is simple and actually very common too, so we're in good company.

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

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

That expression isn't as scary as it first looks. The letter **e** is a mathematical constant 2.71828… It's a very interesting number that pops up in all sorts of areas of mathematics and physics, and the reason I've used the dots … is because the decimal digits keep going on forever. Numbers like that have a fancy name, transcendental numbers. That's all very well and fun, but for our purposes you can just think of it as 2.71828. The input **x** is negated and **e** is raised to the power of that **-x**. The result is added to 1, so we have $\mathbf{1+e^{-x}}$. Finally the inverse is taken of the whole thing, that is 1 is divided by $\mathbf{1+e^{-x}}$. That is what the mildly scary looking function above does to the input **x** to give us the output value **y**. So not so scary after all!

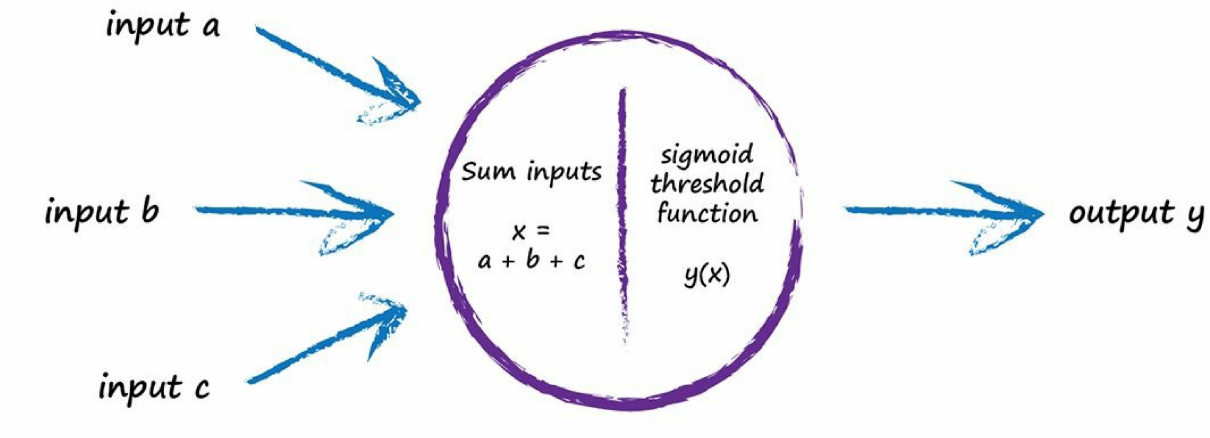
Just out of interest, when **x** is zero, $\mathbf{e^{-x}}$ is 1 because anything raised to a power of zero is 1. So **y** becomes 1 / (1 +1) or simply 1/2, a half. So the basic sigmoid cuts the y-axis at **y** = 1/2.

There is another, very powerful reason for choosing this sigmoid function over the many many other S-shaped functions we could have used for a neuron's output. The reason is that this sigmoid function is much easier to do calculations with than other S-shaped functions, and we'll see why in practice later.

Let's get back to neurons, and consider how we might model an artificial neuron.

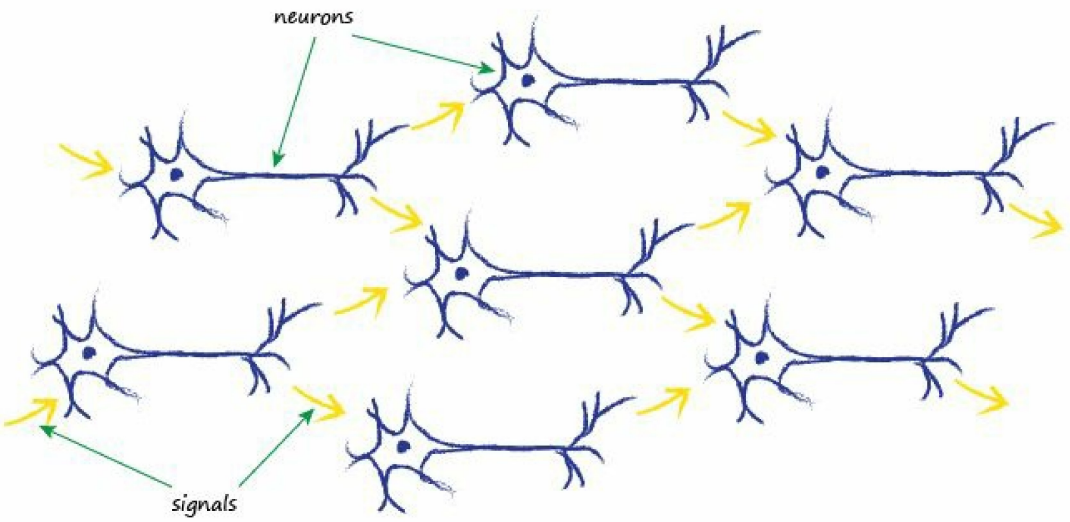
The first thing to realise is that real biological neurons take many inputs, not just one. We saw this when we had two inputs to the Boolean logic machine, so the idea of having more than one input is not new or unusual.

What do we do with all these inputs? We simply combine them by adding them up, and the resultant sum is the input to the sigmoid function which controls the output. This reflects how real neurons work. The following diagram illustrates this idea of combining inputs and then applying the threshold to the combined sum:



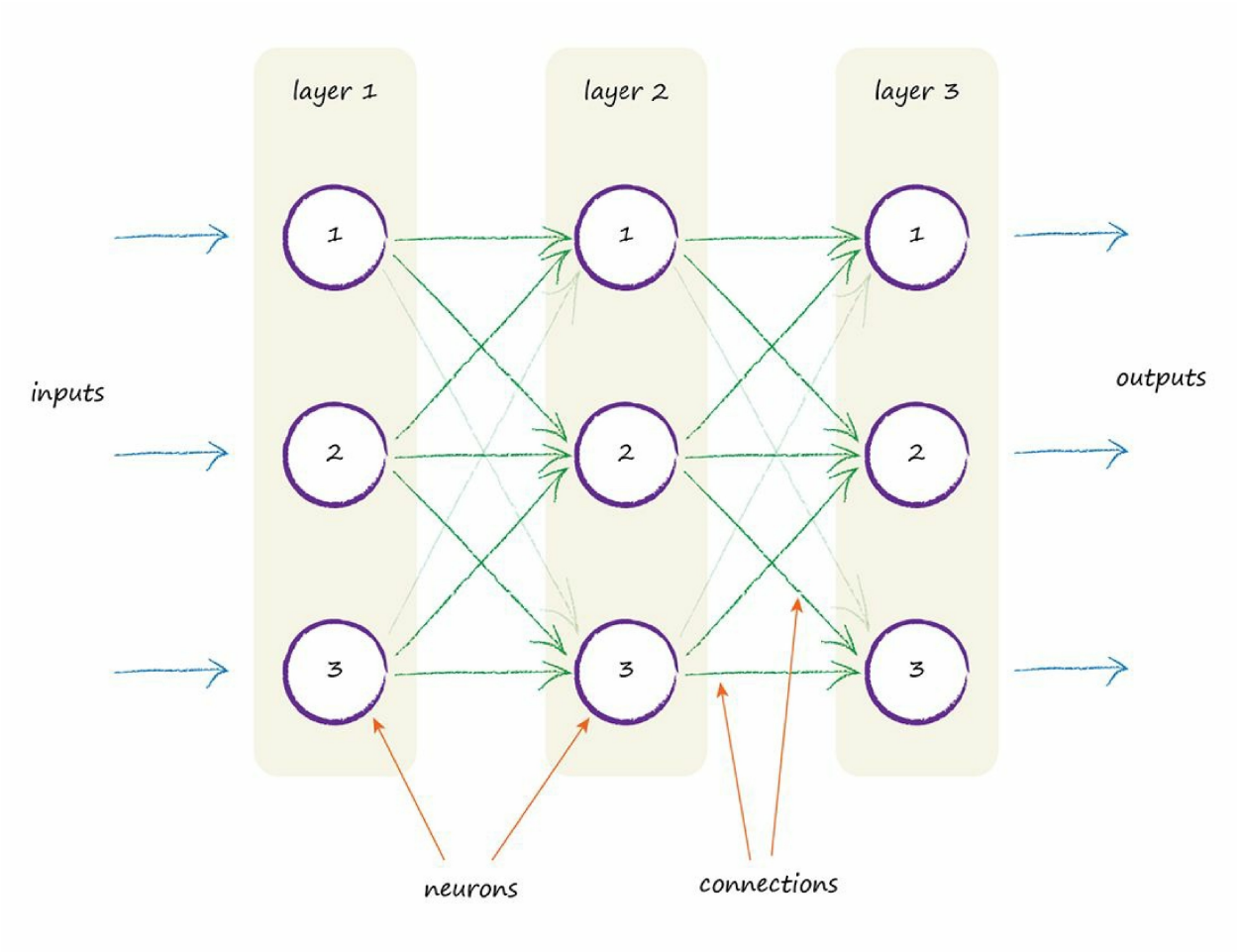
If the combined signal is not large enough then the effect of the sigmoid threshold function is to suppress the output signal. If the sum **x** if large enough the effect of the sigmoid is to fire the neuron. Interestingly, if only one of the several inputs is large and the rest small, this may be enough to fire the neuron. What's more, the neuron can fire if some of the inputs are individually almost, but not quite, large enough because when combined the signal is large enough to overcome the threshold. In an intuitive way, this gives you a sense of the more sophisticated, and in a sense fuzzy, calculations that such neurons can do.

The electrical signals are collected by the dendrites and these combine to form a stronger electrical signal. If the signal is strong enough to pass the threshold, the neuron fires a signal down the axon towards the terminals to pass onto the next neuron's dendrites. The following diagram shows several neurons connected in this way:



The thing to notice is that each neuron takes input from many before it, and also provides signals to many more, if it happens to be firing.

One way to replicate this from nature to an artificial model is to have layers of neurons, with each connected to every other one in the preceding and subsequent layer. The following diagram illustrates this idea:

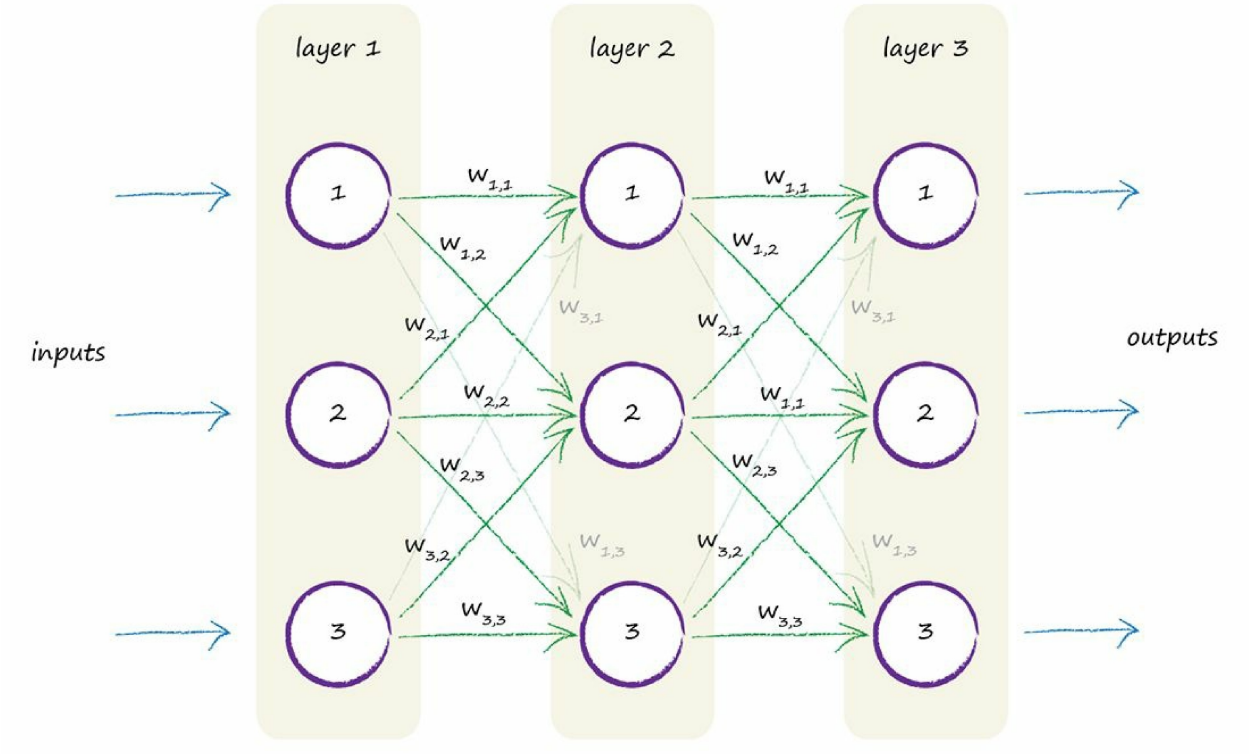


You can see the three layers, each with three artificial neurons, or **nodes**. You can also see each node connected to every other node in the preceding and next layers.

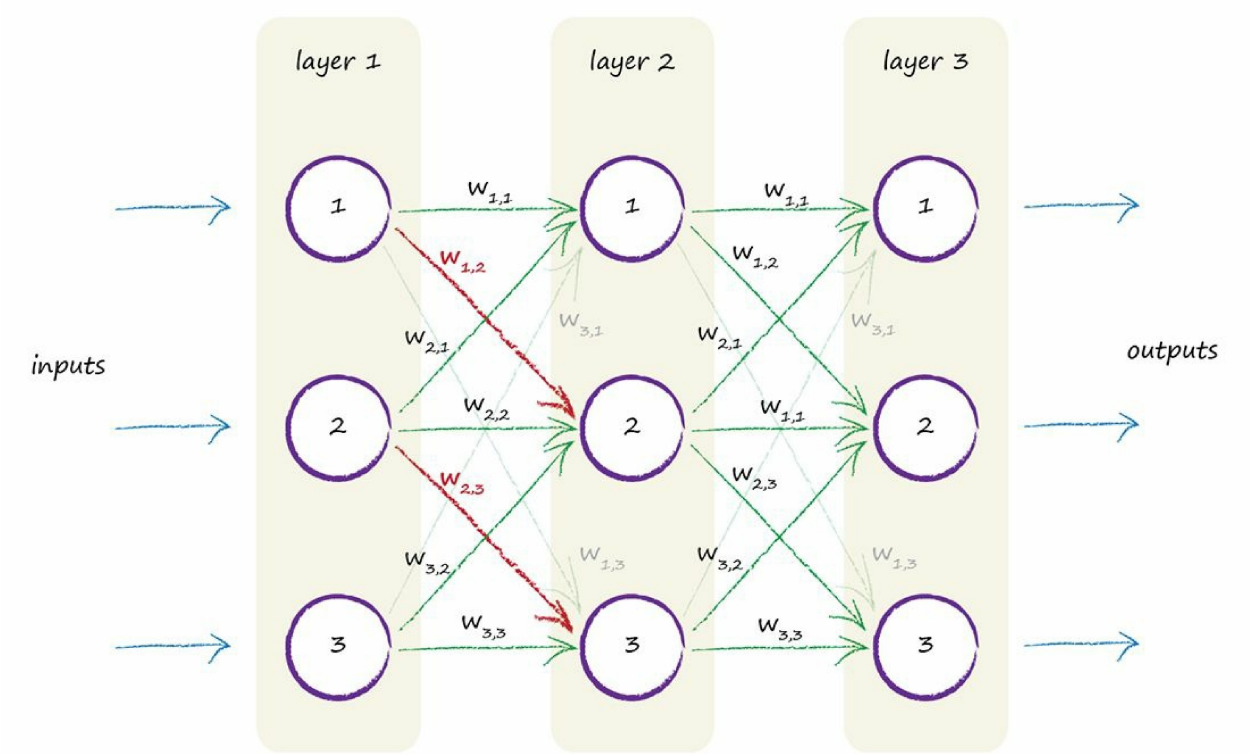
That's great! But what part of this cool looking architecture does the learning? What do we adjust in response to training examples? Is there a parameter that we can refine like the slope of the linear classifiers we looked at earlier?

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.

If the simpler approach works, let's stick with it! 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.



It's worth explaining the funny little numbers next to the weight symbols. The weight $\mathbf{w_{2,3}}$ is simply the weight associated with the signal that passed between node 2 in a layer to node 3 in the next layer. So $\mathbf{w_{1,2}}$ is the weight that diminishes or amplifies the signal between node 1 and node 2 in the next layer. To illustrate the idea, the following diagram shows these two connections between the first and second layer highlighted.



You might reasonably challenge this design and ask yourself why each node should connect to every other node in the previous and next layer. They don't have to and you could connect them in all sorts of creative ways. We don't because the uniformity of this full connectivity is actually easier to encode as computer instructions, and because there shouldn't be any big harm in having a few more

connections than the absolute minimum that might be needed for solving a specific task. 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.

Key Points:

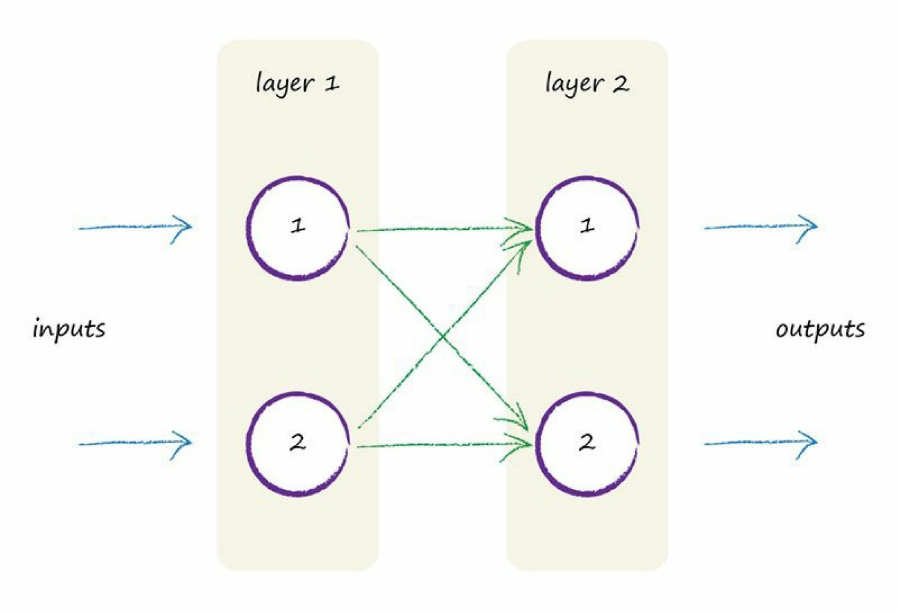
- Biological brains seem to perform sophisticated tasks like flight, finding food, learning language, and evading predators, despite appearing to have much less storage, and running much slower, than modern computers.
- Biological brains are also incredibly resilient to damage and imperfect signals compared to traditional computer systems.
- Biological brains, made of connected neurons, are the inspiration for artificial neural networks.

Following Signals Through A Neural Network

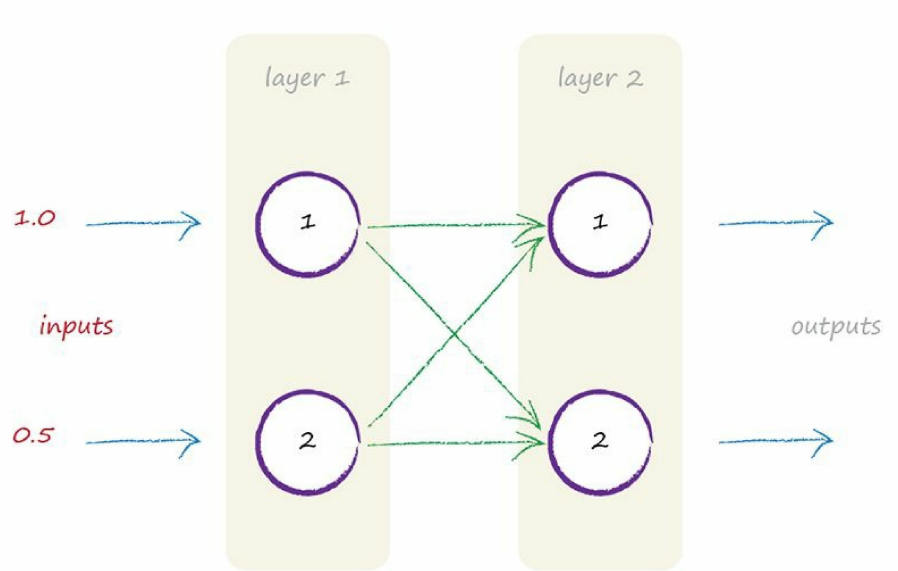
That picture of 3 layers of neurons, with each neuron connect to every other in the next and previous layers, looks pretty amazing.

But the idea of calculating how signals progress from the inputs through the layers to become the outputs seems a little daunting and, well, too much like hard work!

I'll agree that it is hard work, but it is also important to illustrate it working so we always know what is really happening inside a neural network, even if we later use a computer to do all the work for us. So we'll try doing the workings out with a smaller neural network with only 2 layers, each with 2 neurons, as shown below:



Let's imagine the two inputs are 1.0 and 0.5. The following shows these inputs entering this smaller neural network.



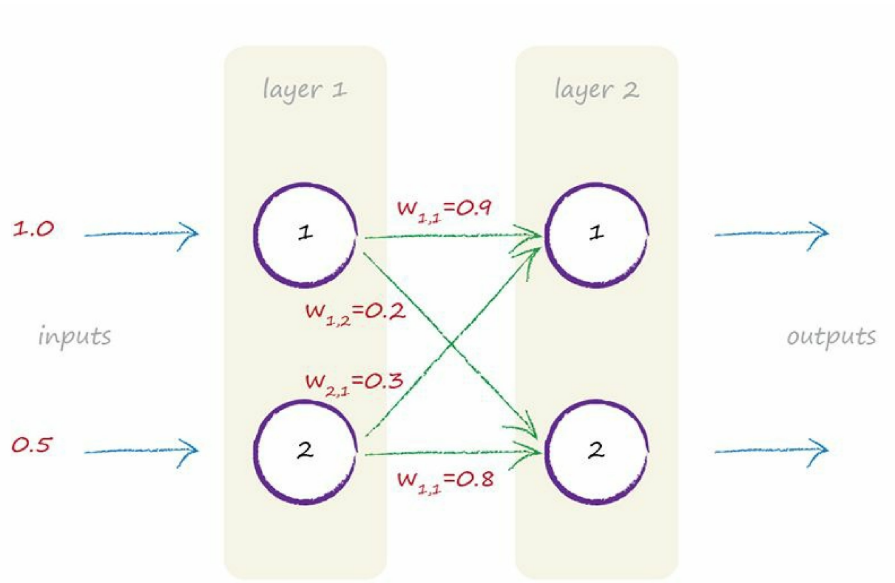
Just as before, each node turns the sum of the inputs into an output using an activation function. We'll also use the sigmoid function $y = \frac{1}{(1+e^{-x})}$ that we saw before, where x is the sum of incoming signals to a neuron, and **y** is the output of that neuron.

What about the weights? That's a very good question - what value should they start with? Let's go with some random weights:

- $\mathbf{w_{1,1}} = 0.9$
- $\mathbf{w_{1,2}} = 0.2$
- $\mathbf{w_{2,1}} = 0.3$
- $\mathbf{w_{2,2}} = 0.8$

Random starting values aren't such a bad idea, and it is what we did when we chose an initial slope value for the simple linear classifiers earlier on. The random value got improved with each example that the classifier learned from. The same should be true for neural networks link weights.

There are only four weights in this small neural network, as that's all the combinations for connecting the 2 nodes in each layer. The following diagram shows all these numbers now marked.



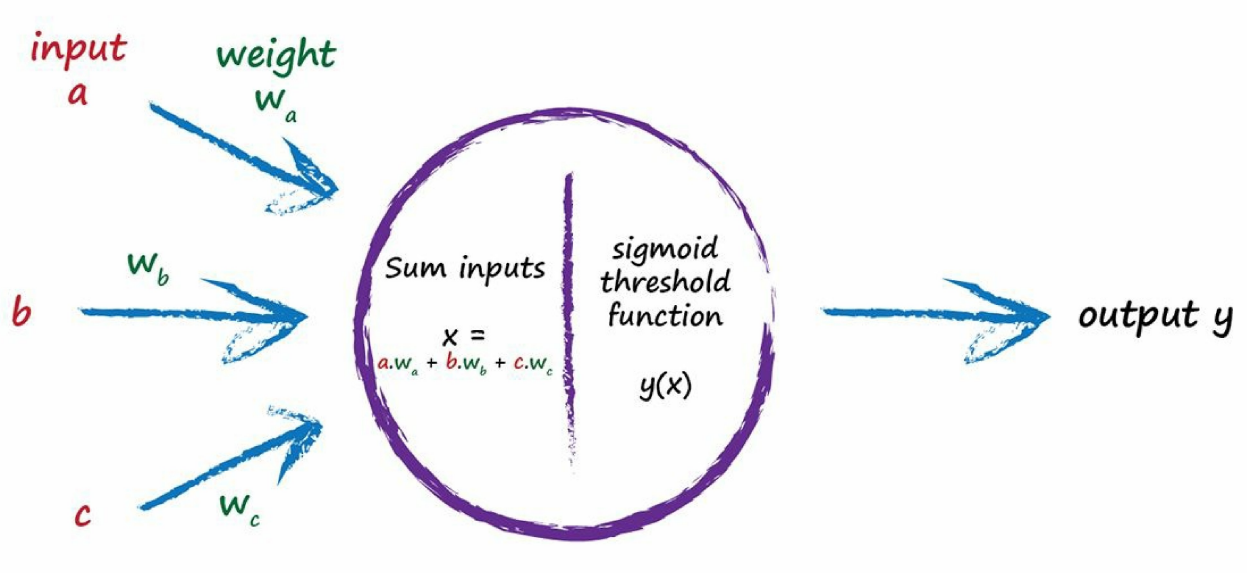
Let's start calculating!

The first layer of nodes is the input layer, and it doesn't do anything other than represent the input signals. That is, the input nodes don't apply an activation function to the input. There is not really a fantastic reason for this other than that's how history took its course. The first layer of neural networks is the input layer and all that layer does is represent the inputs - that's it.

The first input layer 1 was easy - no calculations to be done there.

Next is the second layer where we do need to do some calculations. For each node in this layer we need to work out the combined input. Remember that sigmoid function, $y = \frac{1}{(1+e^{-x})}$? Well, the x in

that function is the combined input into a node. That combination was the raw outputs from the connected nodes in the previous layer, but moderated by the link weights. The following diagram is like the one we saw previously but now includes the need to moderate the incoming signals with the link weights.



So let's first focus on node 1 in the layer 2. Both nodes in the first input layer are connected to it. Those input nodes have raw values of 1.0 and 0.5. The link from the first node has a weight of 0.9 associated with it. The link from the second has a weight of 0.3. So 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

If we didn't moderate the signal, we'd have a very simple addition of the signals 1.0 + 0.5, but we don't want that. It is the weights that do the learning in a neural networks as they are iteratively refined to give better and better results.

So, we've now got **x** = 1.05 for the combined moderated input into the first node of the second layer.

We can now, finally, calculate that node's output using the activation function $y = \frac{1}{(1+e^{-x})}$. Feel free to use a calculator to do this. The answer is **y** = 1 / (1 + 0.3499) = 1 / 1.3499. So **y** = 0.7408.

That's great work! We now have an actual output from one of the network's two output nodes.

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)

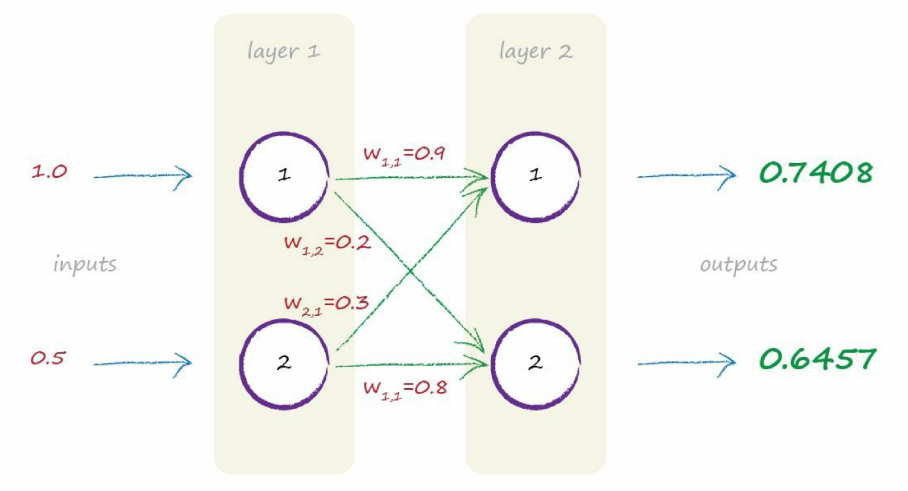
x = (1.0 * 0.2) + (0.5 * 0.8)

$$\mathbf{x} = 0.2 + 0.4$$

$$\mathbf{x} = 0.6$$

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:



That was a fair bit of work just to get two outputs from a very simplified network. I wouldn't want to do the calculations for a larger network by hand at all! Luckily computers are perfect for doing lots of calculations quickly and without getting bored.

Even so, I don't want to write out computer instructions for doing the calculation for a network with more than 2 layers, and with maybe 4, 8 or even 100s of nodes in each layer. Even writing out the instructions would get boring and I'd very likely make mistakes writing all those instructions for all those nodes and all those layers, never mind actually doing the calculations.

Luckily mathematics helps us be extremely concise in writing down the calculations needed to work out the output from a neural network, even one with many more layers and nodes. This conciseness is not just good for us human readers, but also great for computers too because the instructions are much shorter and the execution is much more efficient too.

This concise approach uses **matrices**, which we'll look at next.

Matrix Multiplication is Useful .. Honest!

Matrices have a terrible reputation. They evoke memories of teeth-grindingly boring laborious and apparently pointless hours spent at school doing matrix multiplication.

Previously we manually did the calculations for a 2-layer network with just 2 nodes in each layer. That was enough work, but imagine doing the same for a network with 5 layers and 100 nodes in each? Just writing out all the necessary calculations would be a huge task … all those combinations of combining signals, multiplied by the right weights, applying the sigmoid activation function, for each node, for each layer … argh!

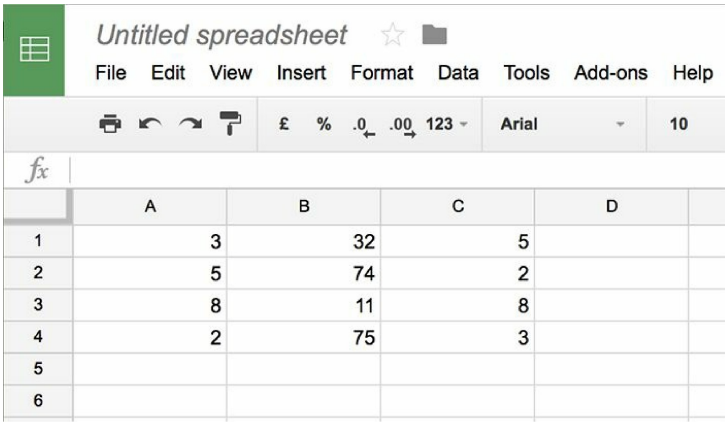
So how can matrices help? Well, they help us in two ways. First, they allow us to compress writing all those calculations into a very simple short form. That's great for us humans, because we don't like doing a lot of work because it's boring, and we're prone to errors anyway. The second benefit is that many computer programming languages understand working with matrices, and because the real work is repetitive, they can recognise that and do it very quickly and efficiently.

In short, matrices allow us to express the work we need to do concisely and easily, and computers can get the calculations done quickly and efficiently.

Now we know why we're going to look at matrices, despite perhaps a painful experience with them at school, let's make a start and demystify them.

A **matrix** is just a table, a rectangular grid, of numbers. That's it. There's nothing much more complex about a matrix than that.

If you've used spreadsheets, you're already comfortable with working with numbers arranged in a grid. Some would call it a table. We can call it a matrix too. The following shows a spreadsheet with a table of numbers.



	A	B	C	D
1	3	32	5	
2	5	74	2	
3	8	11	8	
4	2	75	3	
5				
6				

That's all a matrix is - a table or a grid of numbers - just like the following example of a matrix of size "2 by 3".

$$\begin{pmatrix} 23 & 43 & 22 \\ 43 & 12 & 54 \end{pmatrix}$$

It is convention to use rows first then columns, so this isn't a "3 by 2" matrix, it is a "2 by 3" matrix.

Also, some people use square brackets around matrices, and others use round brackets like we have.

Actually, they don't have to be numbers, they could be quantities which we give a name to, but may not have assigned an actual numerical value to. So the following is a matrix, where each element is a **variable** which has a meaning and could have a numerical value, but we just haven't said what it is yet.

$$\begin{pmatrix} \text{longitude of ship} & \text{longitude of plane} \\ \text{lattitude of ship} & \text{lattitude of plane} \end{pmatrix}$$

Now, matrices become useful to us when we look at how they are multiplied. You may remember how to do this from school, but if not we'll look at it again.

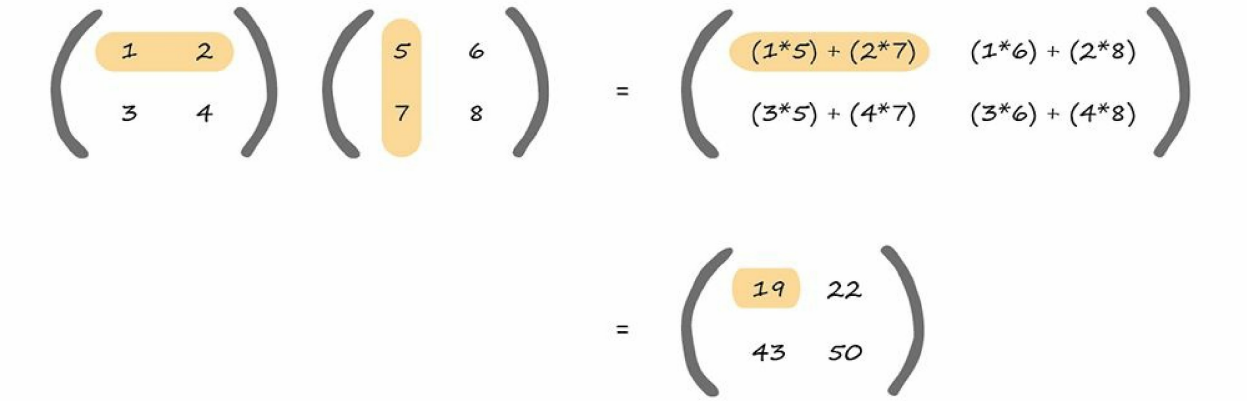
Here's an example of two simple matrices multiplied together.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5)+(2*7) & (1*6)+(2*8) \\ (3*5)+(4*7) & (3*6)+(4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

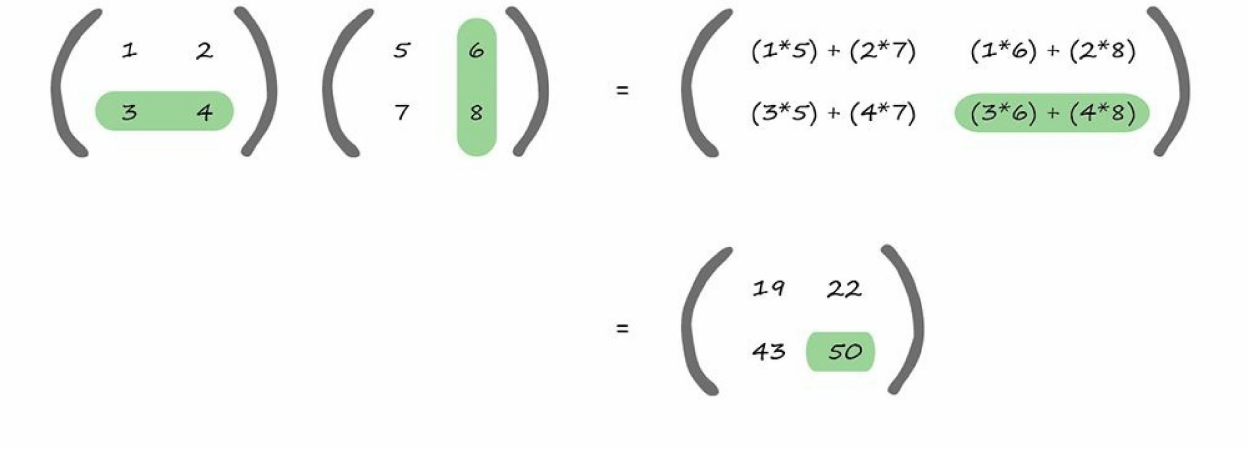
You can see that we don't simply multiply the corresponding elements. The top left of the answer isn't 1*5, and the bottom right isn't 4*8.

Instead, matrices are multiplied using different rules. You may be able to work them out by looking at the above example. If not, have a look at the following which highlights how the top left element of the answer is worked out.



You can see the top left element is worked out by following the top row of the first matrix, and the left column of the second matrix. As you follow these rows and columns, you multiply the elements you encounter and keep a running total. So to work out the top left element of the answer we start to move along the top row of the first array where we find the number 1, and as we start to move down the left column of the second matrix we find the number 5. We multiply 1 and 5 and keep the answer, which is 5, with us. We continue moving along the row and down the column to find the numbers 2 and 7. Multiplying 2 and 7 gives us 14, which we keep too. We've reached the end of the rows and columns so we add up all the numbers we kept aside, which is 5 + 14, to give us 19. That's the top left element of the result matrix.

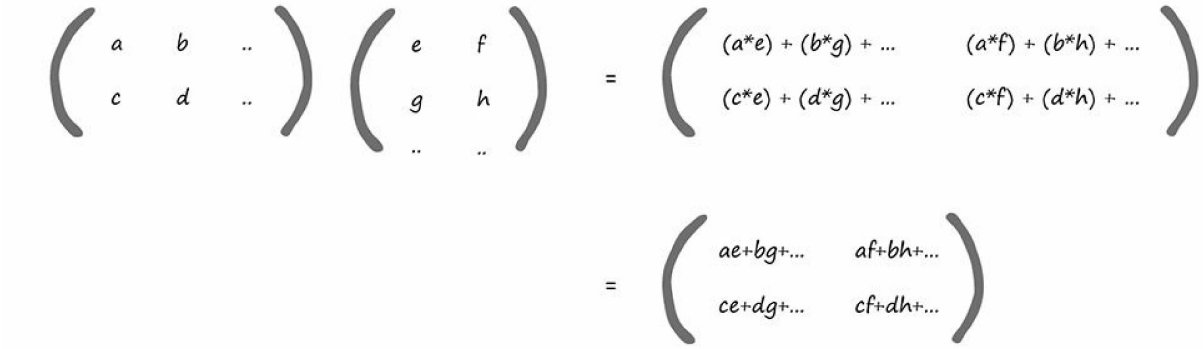
That's a lot of words, but it is easier to see it in action. Have a go yourself. The following explains how the bottom right element is worked out.



You can see again, that following the corresponding row and column of the element we're trying to calculate (the second row and second column, in this example) we have (3*6) and (4*8) which is 18 + 32 = 50.

For completeness, the bottom left is calculated as (3*5) + (4*7) = 15 + 28 = 43. Similarly, the top right is calculated as (1*6) + (2*8) = 6 + 16 = 22.

The following illustrates the rule using variables, rather than numbers.



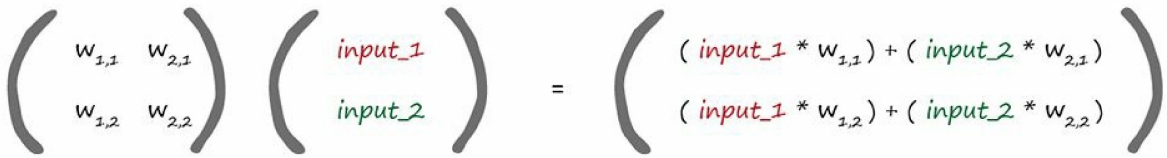
This is just another way of explaining the approach we take to multiplying matrices. By using letters, which could be any number, we've made even clearer the generic approach to multiplying matrices. It's a generic approach because it could be applied to matrices of different sizes too.

When we said it works for matrices of different sizes, there is an important limit. You can't just multiply any two matrices, they need to be compatible. You may have seen this already as you followed the rows across the first matrix, and the columns down the second. If the number of elements in the rows don't match the number of elements in the columns then the method doesn't work. So you can't multiply a "2 by 2" matrix by a "5 by 5" matrix. Try it - you'll see why it doesn't work. To multiply matrices the number of columns in the first must be equal to the number of rows in the second.

In some guides, you'll see this kind of matrix multiplication called a **dot product** or an **inner product**. There are actually different kinds of multiplication possible for matrices, such as a cross product, but the dot product is the one we want here.

Why have we gone down what looks like a rabbit hole of dreaded matrix multiplication and distasteful algebra? There is a very good reason .. hang in there!

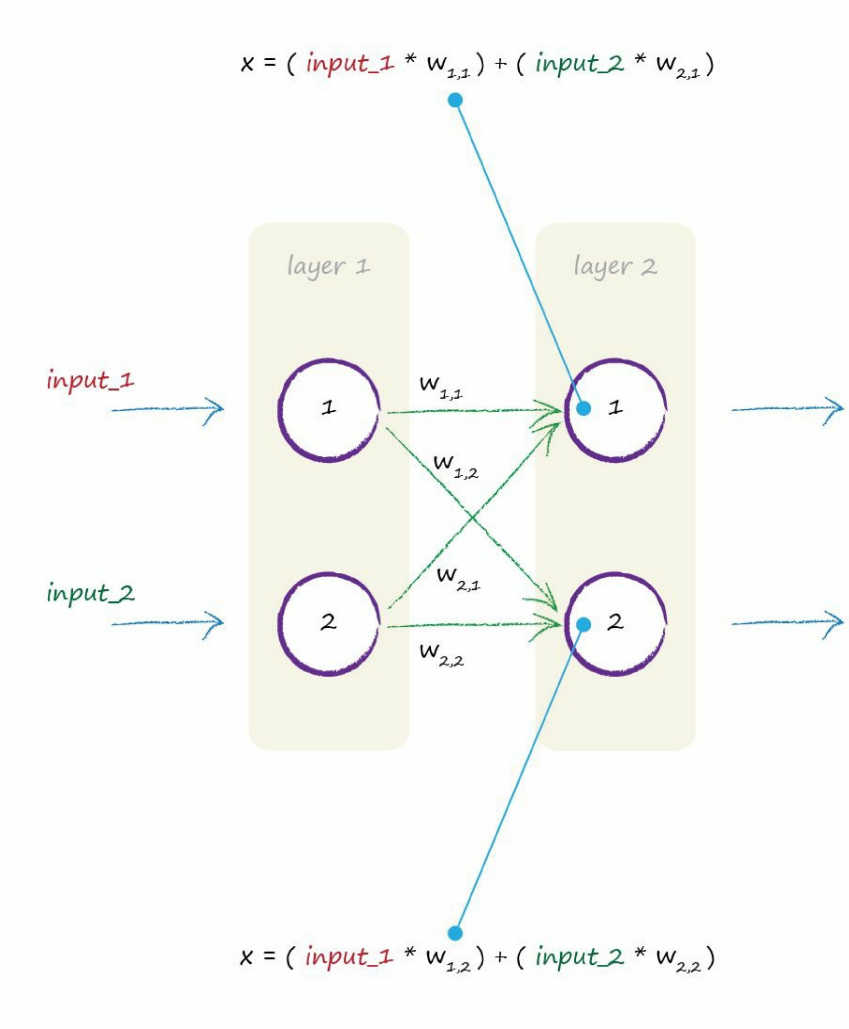
Look what happens if we replace the letters with words that are more meaningful to our neural networks. The second matrix is a two by one matrix, but the multiplication approach is the same.



Magic!

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. Look carefully, and you'll see this. The first node has the first input_1 moderated by the weight $\mathbf{w_{1,1}}$ added to the second input_2 moderated by the weight $\mathbf{w_{2,1}}$. These are the values of **x** before the sigmoid activation function is applied.

The following diagram shows this even more clearly.



This is really very useful!

Why? 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:

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

That is, **W** is the matrix of weights, **I** is the matrix of inputs, and **X** is the resultant matrix of combined moderated signals into layer 2. Matrices are often written in **bold** to show that they are in fact matrices and don't just represent single numbers.

We now don't need to care so much about how many nodes there are in each layer. If we have more nodes, the matrices will just be bigger. But we don't need to write anything longer or larger. We can simply write **W.I** even if I has 2 elements or 200 elements!

Now, if a computer programming language can understand matrix notation, it can do all the hard work of many calculations to work out the **X = W.I**, without us having to give it all the individual instructions for each node in each layer.

This is fantastic! A little bit of effort to understand matrix multiplication has given us a powerful tool for implementing neural networks without lots of effort from us.

What about the activation function? That's easy and doesn't need matrix multiplication. All we need to do is apply the sigmoid function $y = \frac{1}{(1+e^{-x})}$ to each individual element of the matrix **X**.

This sounds too simple, but it is correct because we're not combining signals from different nodes here, we've already done that and the answers are in **X**. As we saw earlier, the activation function simply applies a threshold and squishes the response to be more like that seen in biological neurons. So the final output from the second layer is:

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

That **O** written in bold is a matrix, which contains all the outputs from the final layer of the neural network.

The expression $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$ applies to the calculations between one layer and the next. If we have 3 layers, for example, we simply do the matrix multiplication again, using the outputs of the second layer as inputs to the third layer but of course combined and moderated using more weights.

Enough theory - let's see how it works with a real example, but this time we'll use a slightly larger neural network of 3 layers, each with 3 nodes.

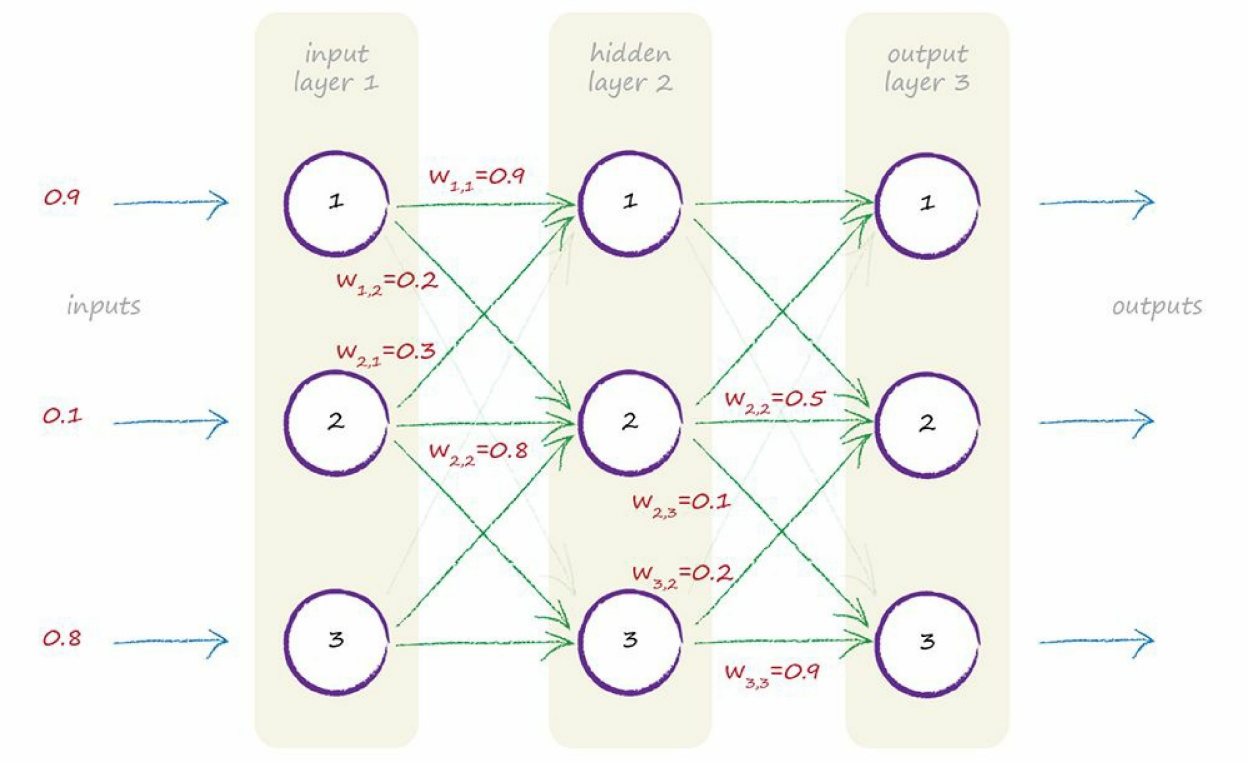
Key Points:

- The many calculations needed to feed a signal forward through a neural network can be expressed as **matrix multiplication**.
- Expressing it as matrix multiplication makes it much more **concise** for us to write, no matter the size of neural network.
- More importantly, some computer programming languages understand matrix calculations, and recognise that the underlying calculations are very similar. This allows them to do these calculations more **efficiently** and quickly.

A Three Layer Example with Matrix Multiplication

We haven't worked through feeding signals through a neural network using matrices to do the calculations. We also haven't worked through an example with more than 2 layers, which is interesting because we need to see how we treat the outputs of the middle layer as inputs to the final third layer.

The following diagram shows an example neural network with 3 layers, each with 3 nodes. To keep the diagram clear, not all the weights are marked.



We'll introduce some of the commonly used terminology here too. The first layer is the **input layer**, as we know. The final layer is the **output layer**, as we also know. The middle layer is called the **hidden layer**. That sounds mysterious and dark, but sadly there isn't a mysterious dark reason for it. The name just stuck because the outputs of the middle layer are not necessarily made apparent as outputs, so are "hidden". Yes, that's a bit lame, but there really isn't a better reason for the name.

Let's work through that example network, illustrated in that diagram. We can see the three inputs are 0.9, 0.1 and 0.8. So the input matrix **I** is:

$$I = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

That was easy. That's the first input layer done, because that's all the input layer does - it merely represents the input.

Next is the middle hidden layer. Here we need to work out the combined (and moderated) signals to each node in this middle layer. Remember each node in this middle hidden layer is connected to by every node in the input layer, so it gets some portion of each input signal. We don't want to go through the many calculations like we did earlier, we want to try this matrix method.

As we just saw, the combined and moderated inputs into this middle layer are $\mathbf{X} = \mathbf{W.I}$ where I is the matrix of input signals, and **W** is the matrix of weights. We have **I** but what is **W**? Well the diagram shows some of the (made up) weights for this example but not all of them. The following shows all of them, again made up randomly. There's nothing particularly special about them in this example.

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

You can see the weight between the first input node and the first node of the middle hidden layer is $\mathbf{w_{1,1}} = 0.9$, just as in the network diagram above. Similarly you can see the weight for the link between the second node of the input and the second node of the hidden layer is $\mathbf{w_{2,2}} = 0.8$, as shown in the diagram. The diagram doesn't show the link between the third input node and the first hidden layer node, which we made up as $\mathbf{w_{3,1}} = 0.1$.

But wait - why have we written "input_hidden" next to that **W**? It's because $\mathbf{W_{input\_hidden}}$ are the weights between the input and hidden layers. We need another matrix of weights for the links between the hidden and output layers, and we can call it $\mathbf{W_{hidden\_output}}$.

The following shows this second matrix $\mathbf{W_{hidden\_output}}$ with the weights filled in as before. Again you should be able to see, for example, the link between the third hidden node and the third output node as $\mathbf{w_{3,3}} = 0.9$.

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

Great, we've got the weight matrices sorted.

Let's get on with working out the combined moderated input into the hidden layer. We should also give it a descriptive name so we know it refers to the combined input to the middle layer and not the final layer. Let's call it $\mathbf{X_{hidden}}$.

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

We're not going to do the entire matrix multiplication here, because that was the whole point of using matrices. We want computers to do the laborious number crunching. The answer is worked out as shown below.

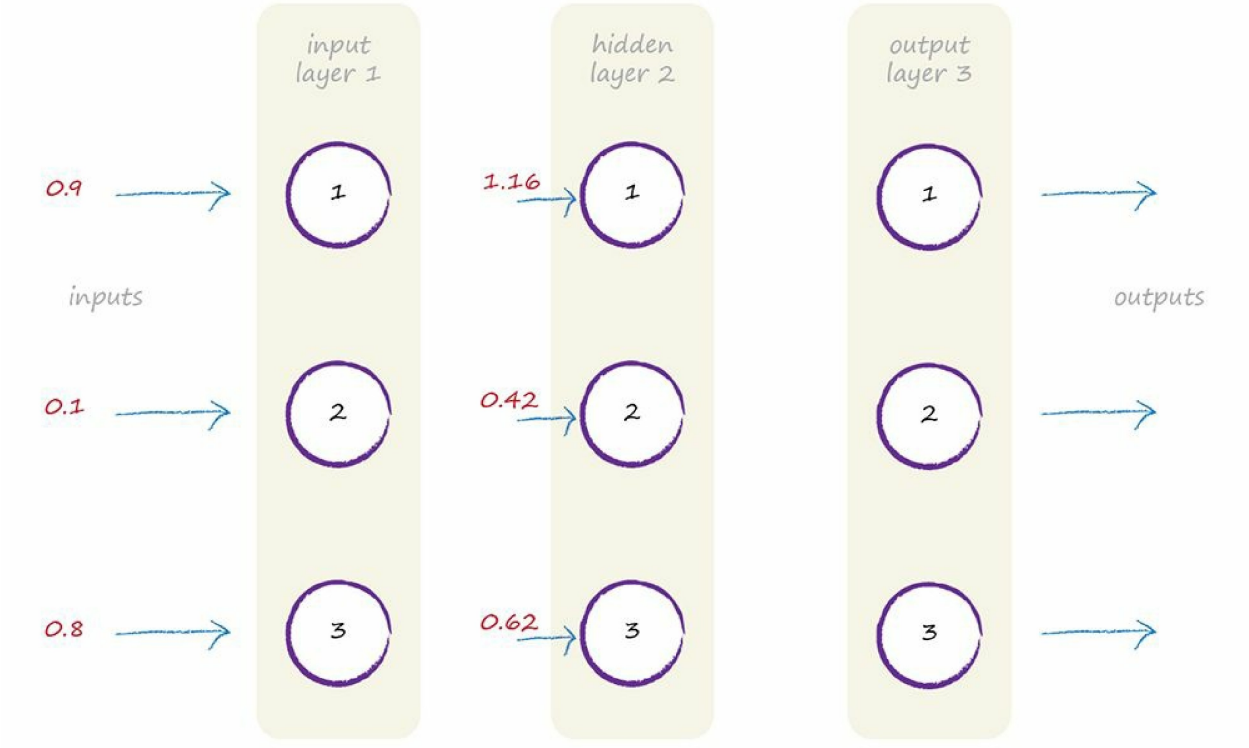
$$X_{hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

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

I used a computer to work this out, and we'll learn to do this together using the Python programming language in part 2 of this guide. We won't do it now as we don't want to get distracted by computer software just yet.

So we have the combined moderated inputs into the middle hidden layer, and they are 1.16, 0.42 and 0.62. And we used matrices to do the hard work for us. That's an achievement to be proud of!

Let's visualise these combined moderated inputs into the second hidden layer.



So far so good, but there's more to do. You'll remember those nodes apply a sigmoid activation function to make the response to the signal more like those found in nature. So let's do that:

$$\mathbf{O_{hidden}} = \text{sigmoid}(\mathbf{X_{hidden}})$$

The sigmoid function is applied to each element in $\mathbf{X_{hidden}}$ to produce the matrix which has the output of the middle hidden layer.

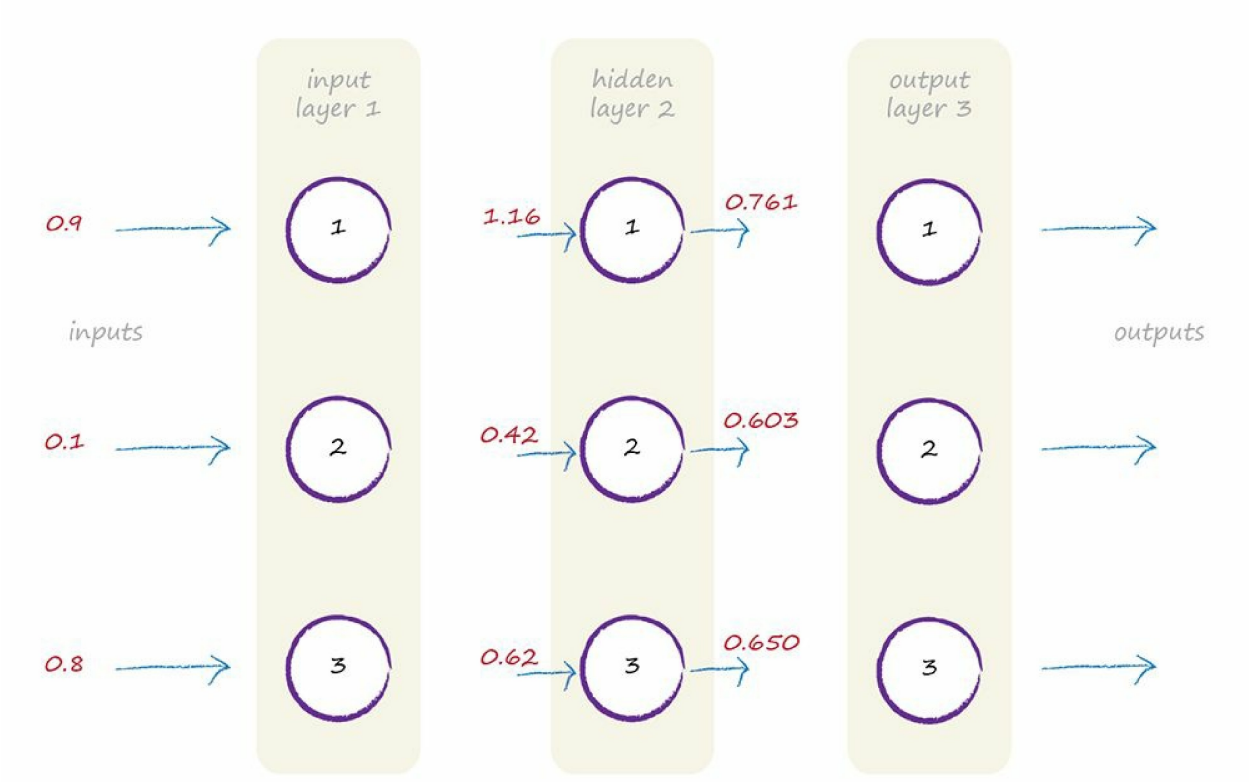
$$O_{hidden} = sigmoid \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

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

Let's just check the first element to be sure. The sigmoid function is $y = \frac{1}{(1+e^{-x})}$, so when x = 1.16, $e^{-1.16}$ is 0.3135. That means **y** = 1/(1+ 0.3135) = 0.761.

You can also see that all the values are between 0 and 1, because this sigmoid doesn't produce values outside that range. Look back at the graph of the logistic function to see this visually.

Phew! Let's pause again and see what we've done. We've worked out the signal as it passes through the middle layer. That is, the outputs from the middle layer. Which, just to be super clear, are the combined inputs into the middle layer which then have the activation function applied. Let's update the diagram with this new information.



If this was a two layer neural network, we'd stop now as these are the outputs from the second layer. We won't stop because we have another third layer.

How do we work out the signal through the third layer? It's the same approach as the second layer, there isn't any real difference. We still have incoming signals into the third layer, just as we did coming into the second layer. We still have links with weights to moderate those signals. And we still have an activation function to make the response behave like those we see in nature. So the thing to remember is, no matter how many layers we have, we can treat each layer like any other - with incoming signals which we combine, link weights to moderate those incoming signals, and an activation function to produce the output from that layer. We don't care whether we're working on the 3rd or 53rd or even the 103rd layer - the approach is the same.

So let's crack on and calculate the combined moderated input into this final layer $\mathbf{X} = \mathbf{W.I}$ just as we did before.

The inputs into this layer are the outputs from the second layer we just worked out $\mathbf{O_{hidden}}$. And the weights are those for the links between the second and third layers $\mathbf{W_{hidden\_output}}$, not those we just used between the first and second. So we have:

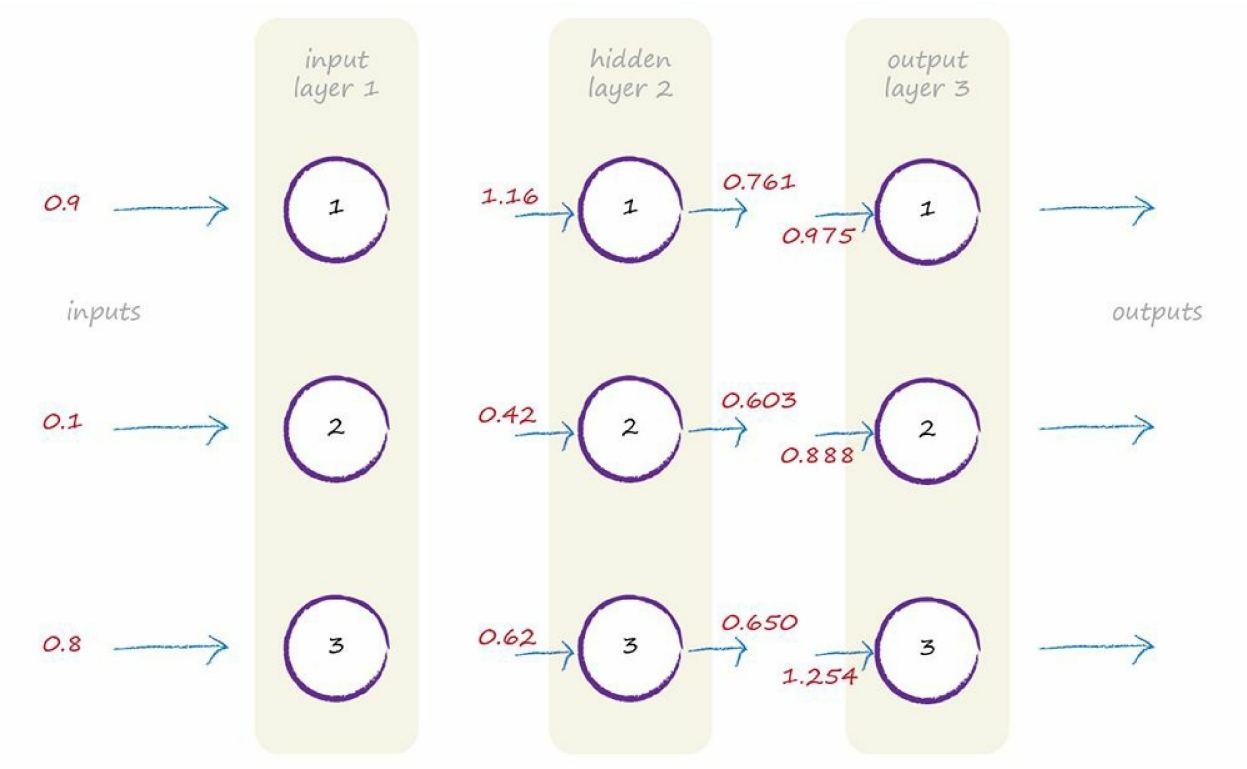
$$\mathbf{X_{output}} = \mathbf{W_{hidden\_output}} \cdot \mathbf{O_{hidden}}$$

So working this out just in the same way gives the following result for the combined moderated inputs into the final output layer.

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

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

The updated diagram now shows our progress feeding forward the signal from the initial input right through to the combined inputs to the final layer.

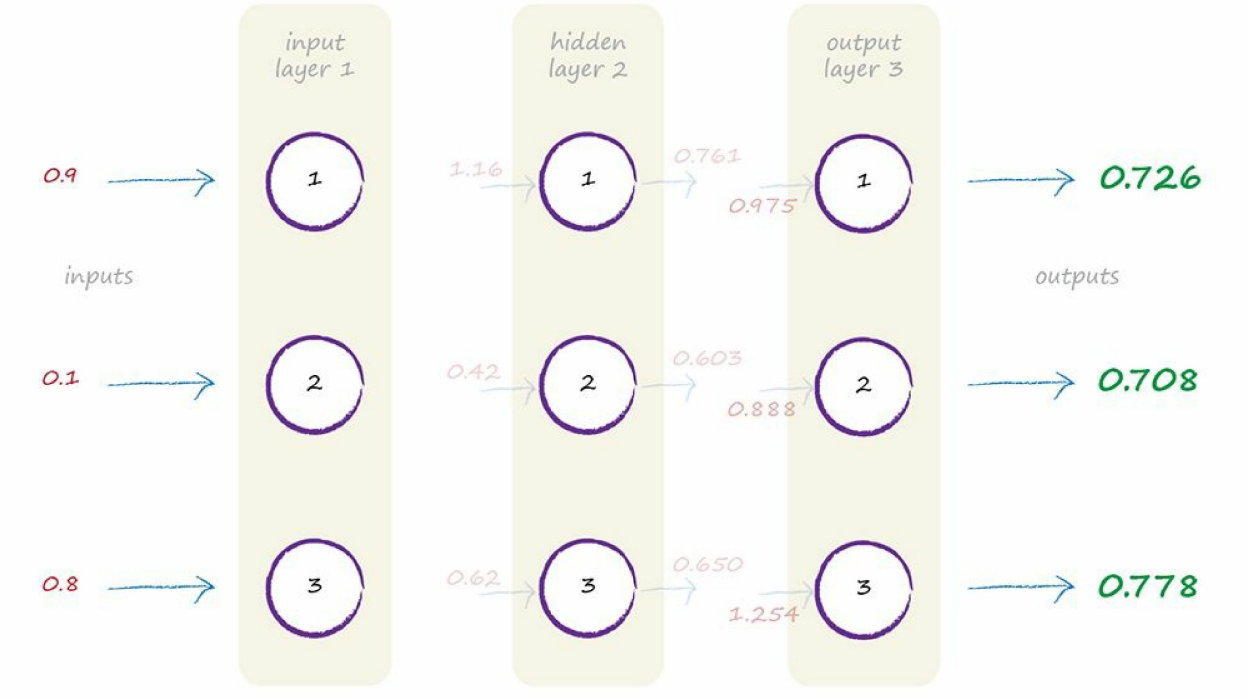


All that remains is to apply the sigmoid activation function, which is easy.

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

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

That's it! We have the final outputs from the neural network. Let's show this on the diagram too.



So the final output of the example neural network with three layers is 0.726, 0.708 and 0.778.

We've successfully followed the signal from it's initial entry into the neural network, through the layers, and out of the final output layer.

What now?

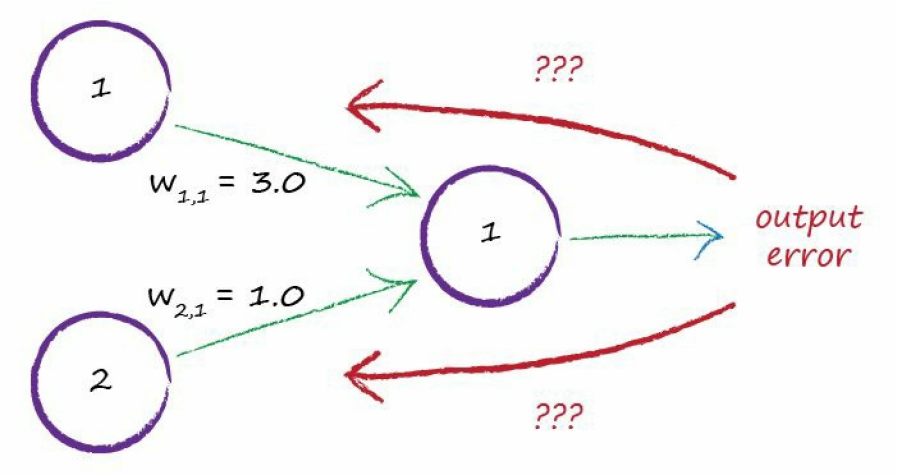
The next step is to use the output from the neural network and compare it with the training example to work out an error. We need to use that error to refine the neural network itself so that it improves its outputs.

This is probably the most difficult thing to understand, so we'll take it gently and illustrate the ideas as we go.

Learning Weights From More Than One Node

Previously we refined a simple linear classifier by adjusting the slope parameter of the node's linear function. 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.

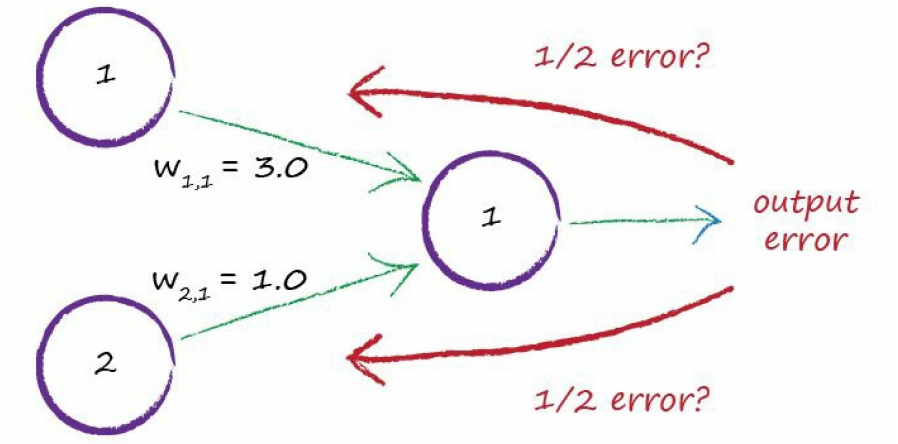


Things were much simpler when we just had one node feeding into an output node. If we have two nodes, how do we use that output error?

It doesn't make sense to use all the error to update only one weight, because that ignores the other link and its weight. That error was there because more than one link contributed to it.

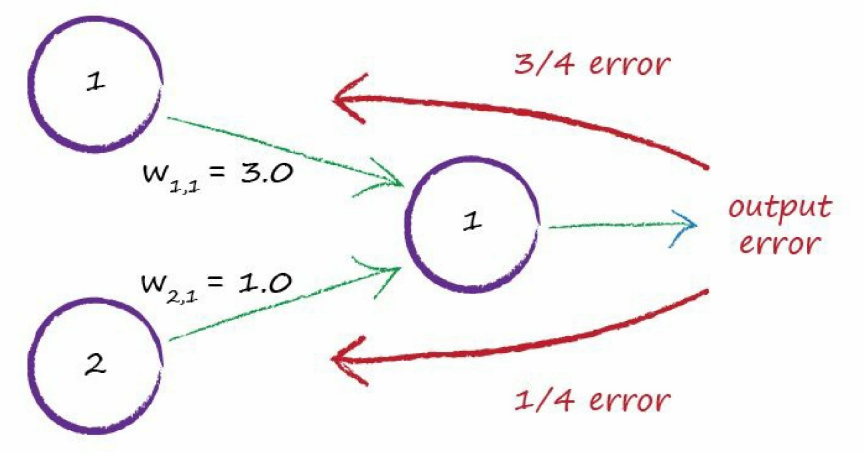
There is a tiny chance that only one link of many was responsible for the error, but that chance is extremely miniscule. If we did change a weight that was already "correct", making it worse, it would get improved during the next iterations so all is not lost.

One idea is to split the error equally amongst all contributing nodes, as shown next.



That is not a bad idea at all. Although I haven't tried it in real neural networks, I'm sure it wouldn't work too badly at all.

Another idea is to split the error but not to do it equally. Instead we give more of the error to those contributing connections which had greater link weights. Why? Because they contributed more to the error. The following diagram illustrates this idea.



Here there are two nodes contributing a signal to the output node. The link weights are 3.0 and 1.0. If we split the error in a way that is proportionate to these weights, we can see that ¾ of the output error should be used to update the first larger weight, and that ¼ of the error for the second smaller weight.

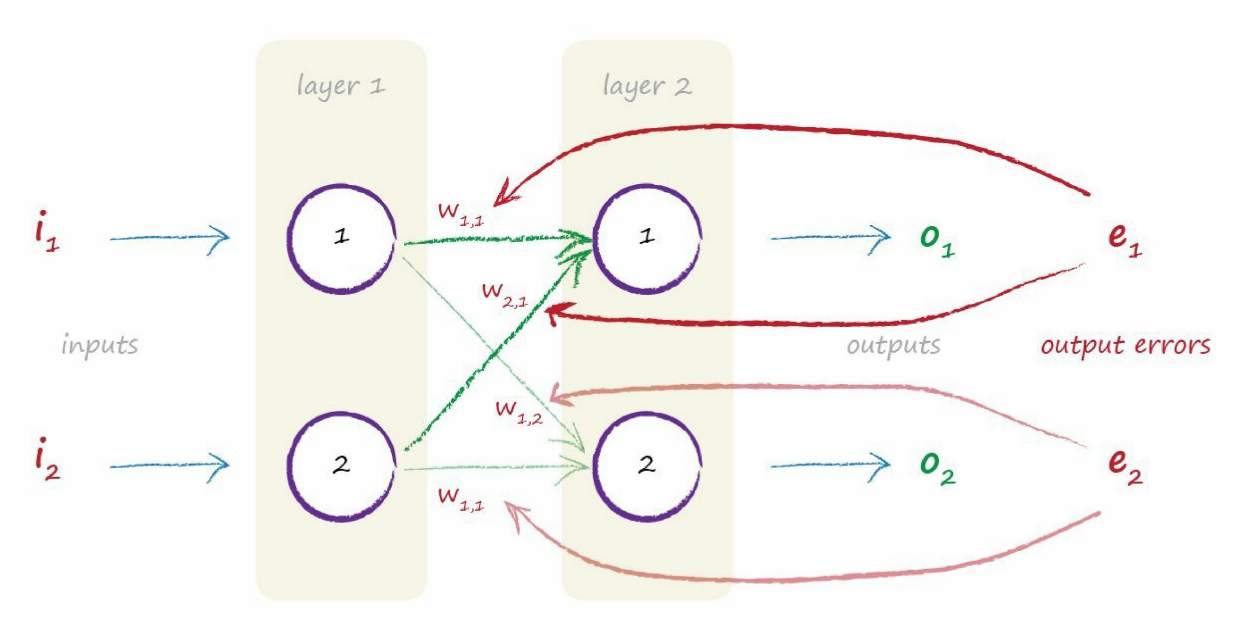
We can extend this same idea to many more nodes. If we had 100 nodes connected to an output node, we'd split the error across the 100 connections to that output node in **proportion** to each link's contribution to the error, indicated by the size of the link's weight.

You can see that we're using the weights in two ways. Firstly we use the weights to propagate signals forward from the input to the output layers in a neural network. We worked on this extensively before. Secondly we use the weights to propagate the error backwards from the output back into the network. You won't be surprised why the method is called **backpropagation**.

If the output layer had 2 nodes, we'd do the same for the second output node. That second output node will have it's own error, which is similarly split across the connecting links. Let's look at this next.

Backpropagating Errors From More Output Nodes

The following diagram shows a simple network with 2 input nodes, but now with 2 output nodes.



Both output nodes can have an error - in fact that's extremely likely when we haven't trained the network. You can see that both of these errors needs to inform the refinement of the internal link weights in the network. We can use the same approach as before, where we split an output node's error amongst the contributing links, in a way that's proportionate to their weights.

The fact that we have more than one output node doesn't really change anything. We simply repeat for the second output node what we already did for the first one. Why is this so simple? It is simple because the links into an output node don't depend on the links into another output node. There is no dependence between these two sets of links.

Looking at that diagram again, we've labelled the error at the first output node as $\mathbf{e_1}$. Remember this is the difference between the desired output provided by the training data $\mathbf{t_1}$ and the actual output $\mathbf{o_1}$. That is, $\mathbf{e_1} = (\mathbf{t_1} - \mathbf{o_1})$. The error at the second output node is labelled $\mathbf{e_2}$.

You can see from the diagram that the error $\mathbf{e_1}$ is split in proportion to the connected links, which have weights $\mathbf{w_{11}}$ and $\mathbf{w_{21}}$. Similarly, $\mathbf{e_2}$ would be split in proportionate to weights $\mathbf{w_{21}}$ and $\mathbf{w_{22}}$.

Let's write out what these splits are, so we're not in any doubt. The error $\mathbf{e_1}$ is used to inform the refinement of both weights $\mathbf{w_{11}}$ and $\mathbf{w_{21}}$. It is split so that the fraction of $\mathbf{e_1}$ used to update $\mathbf{w_{11}}$ is

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

Similarly the fraction of $\mathbf{e_1}$ used to refine $\mathbf{w_{21}}$ is

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

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 $\mathbf{e_1}$ is split to give more to the weight that is larger, and less to the weight that is smaller.

If $\mathbf{w_{11}}$ is twice as large as $\mathbf{w_{21}}$, say $\mathbf{w_{11}}$=6 and $\mathbf{w_{21}}$=3, then the fraction of $\mathbf{e_1}$ used to update $\mathbf{w_{11}}$ is 6/(6+3) = 6/9 = 2/3. That should leave 1/3 of $\mathbf{e_1}$ for the other smaller weight $\mathbf{w_{21}}$ which we can confirm using the expression 3/(6+3) = 3/9 which is indeed 1/3.

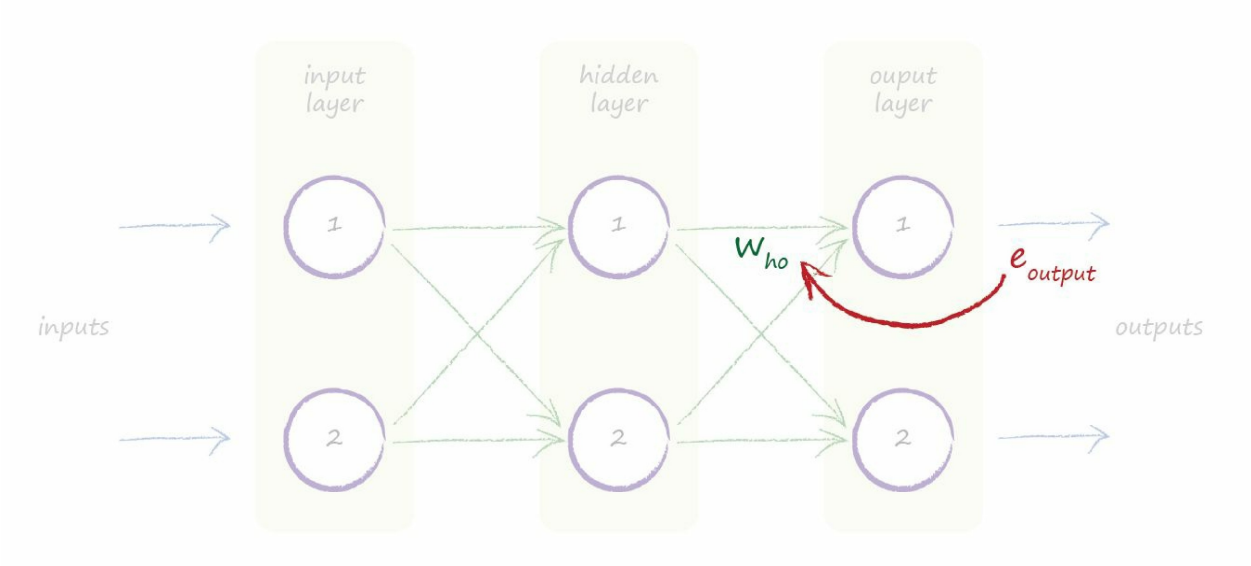
If the weights were equal, the fractions will both be half, as you'd expect. Let's see this just to be sure. Let's say $\mathbf{w_{11}}$=4 and $\mathbf{w_{21}}$=4, then the fraction is 4/(4+4) = 4/8 = 1/2 for both cases.

Before we go further, let's pause and take a step back and see what we've done from a distance. We knew we needed to use the error to guide the refinement of some parameter inside the network, in this case the link weights. We've just seen how to do that for the link weights which moderate signals into the final output layer of a neural network. We've also seen that there isn't a complication when there is more than one output node, we just do the same thing for each output node. Great!

The next question to ask is what happens when we have more than 2 layers? How do we update the link weights in the layers further back from the final output layer?

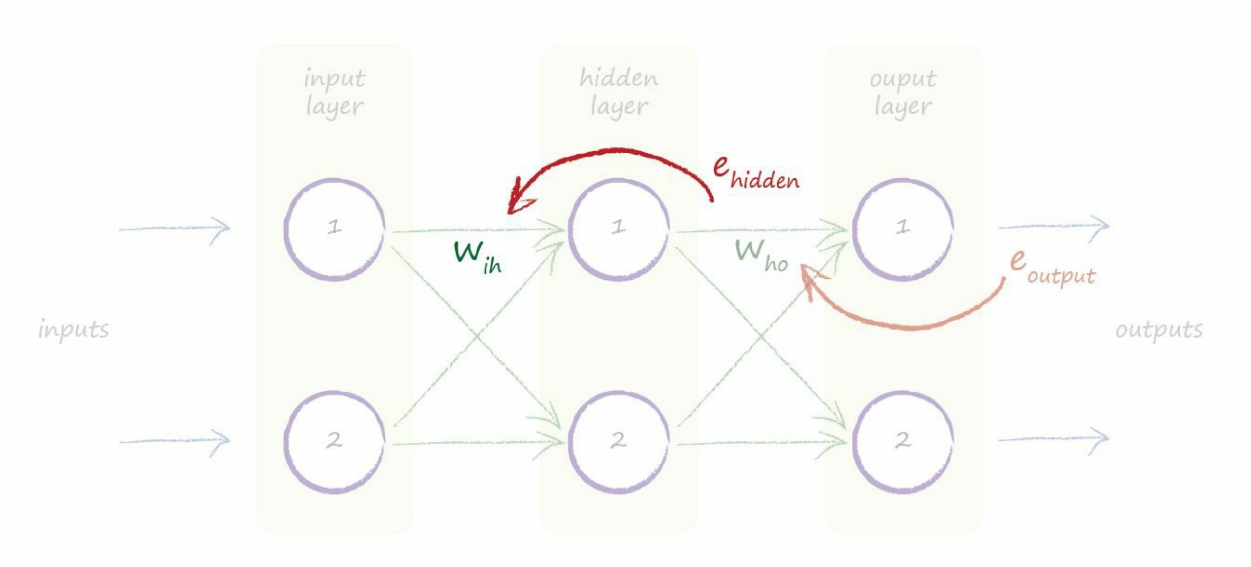
Backpropagating Errors To More Layers

The following diagram shows a simple neural network with 3 layers, an input layer, a hidden layer and the final output layer.



Working back from the final output layer at the right hand side, we can see that we use the errors in that output layer to guide the refinement of the link weights feeding into the final layer. We've labelled the output errors more generically as $\mathbf{e_{output}}$ and the weights of the links between the hidden and output layer as $\mathbf{w_{ho}}$. We worked out the specific errors associated with each link by splitting the weights in proportion to the size of the weights themselves.

By showing this visually, we can see what we need to do for the new additional layer. We simply take those errors associated with the output of the hidden layer nodes $\mathbf{e_{hidden}}$, and split those again proportionately across the preceding links between the input and hidden layers $\mathbf{w_{ih}}$. The next diagram shows this logic.

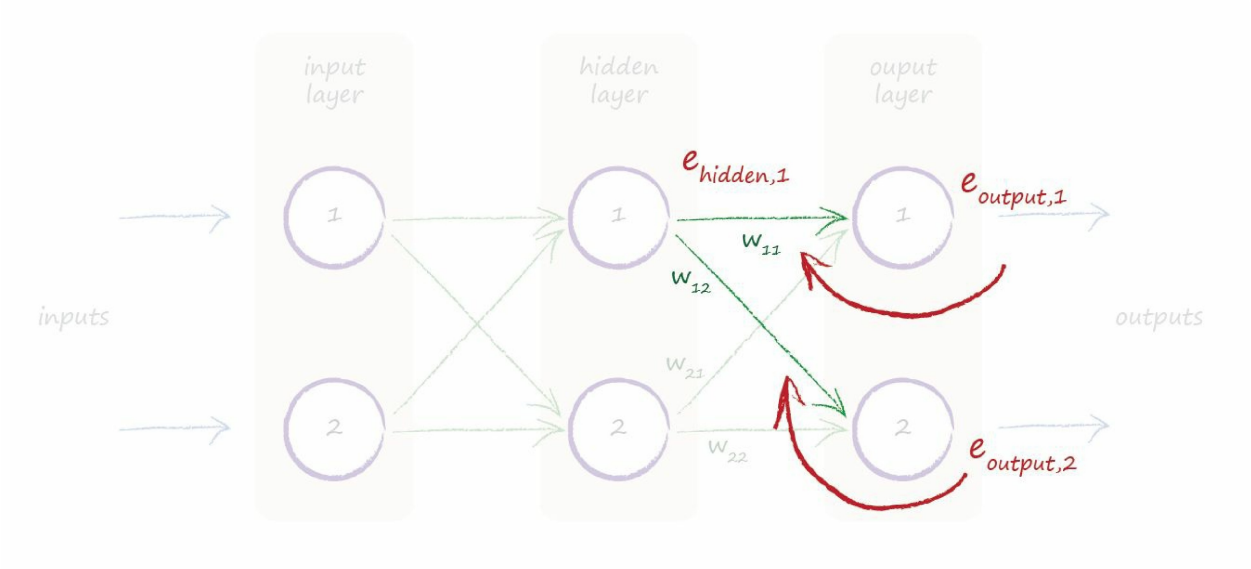


If we had even more layers, we'd repeatedly apply this same idea to each layer working backwards from the final output layer. The flow of error information makes intuitive sense. You can see again

why this is called error **backpropagation**.

If we first used the error in the output of the output layer nodes $\mathbf{e_{output}}$, what error do we use for the hidden layer nodes $\mathbf{e_{hidden}}$? This is a good question to ask because a node in the middle hidden layer doesn't have an obvious error. We know from feeding forward the input signals that, yes, each node in the hidden layer does indeed have a single output. You'll remember that was the activation function applied to the weighted sum on the inputs to that node. But how do we work out the error?

We don't have the target or desired outputs for the hidden nodes. We only have the target values for the final output layer nodes, and these come from the training examples. Let's look at that diagram above again for inspiration! That first node in the hidden layer has two links emerging from it to connect it to the two output layer nodes. We know we can split the output error along each of these links, just as we did before. That means we have some kind of error for each of the two links that emerge from this middle layer node. We could recombine these two link errors to form the error for this node as a second best approach because we don't actually have a target value for the middle layer node. The following shows this idea visually.



You can see more clearly what's happening, but let's go through it again just to be sure. We need an error for the hidden layer nodes so we can use it to update the weights in the preceding layer. We call these $\mathbf{e_{hidden}}$. But we don't have an obvious answer to what they actually are. We can't say the error is the difference between the desired target output from those nodes and the actual outputs, because our training data examples only give us targets for the very final output nodes.

The training data examples only tell us what the outputs from the very final nodes should be. They don't tell us what the outputs from nodes in any other layer should be. This is the core of the puzzle.

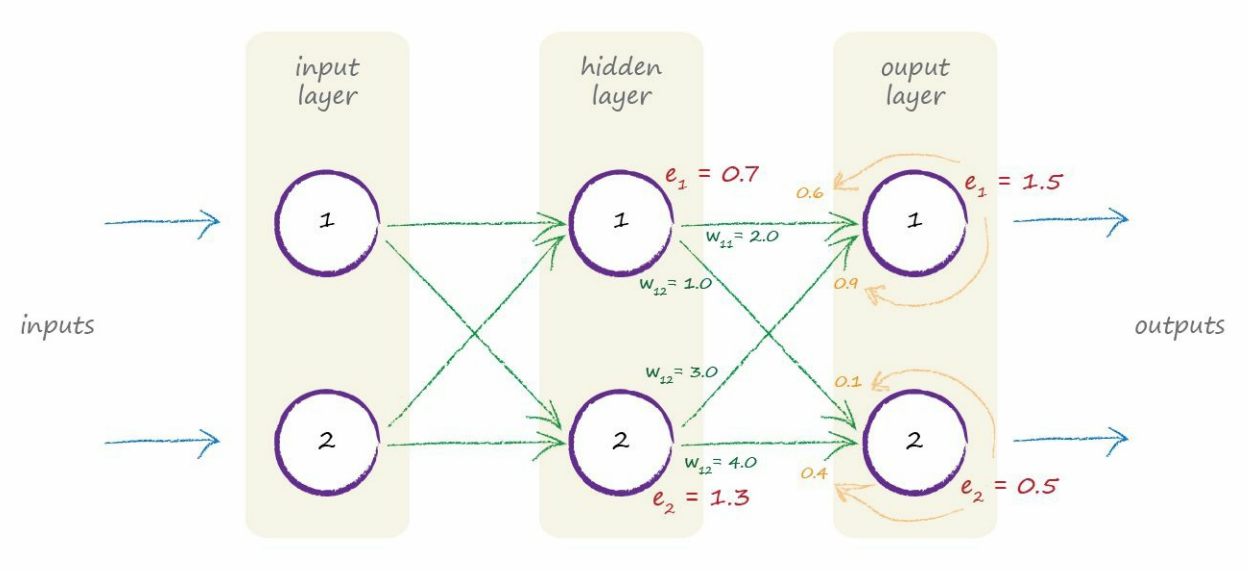
We could recombine the split errors for the links using the error backpropagation we just saw earlier. So the error in the first hidden node is the sum of the split errors in all the links connecting forward from same node. In the diagram above, we have a fraction of the output error $\mathbf{e_{output,1}}$ on the link with weight $\mathbf{w_{11}}$ and also a fraction of the output error $\mathbf{e_{output,2}}$ from the second output node on the link with weight $\mathbf{w_{12}}$.

So let's write this down.

$$e_{hidden,1} = \text{sum of split errors on links } w_{11} \text{ and } w_{12}$$

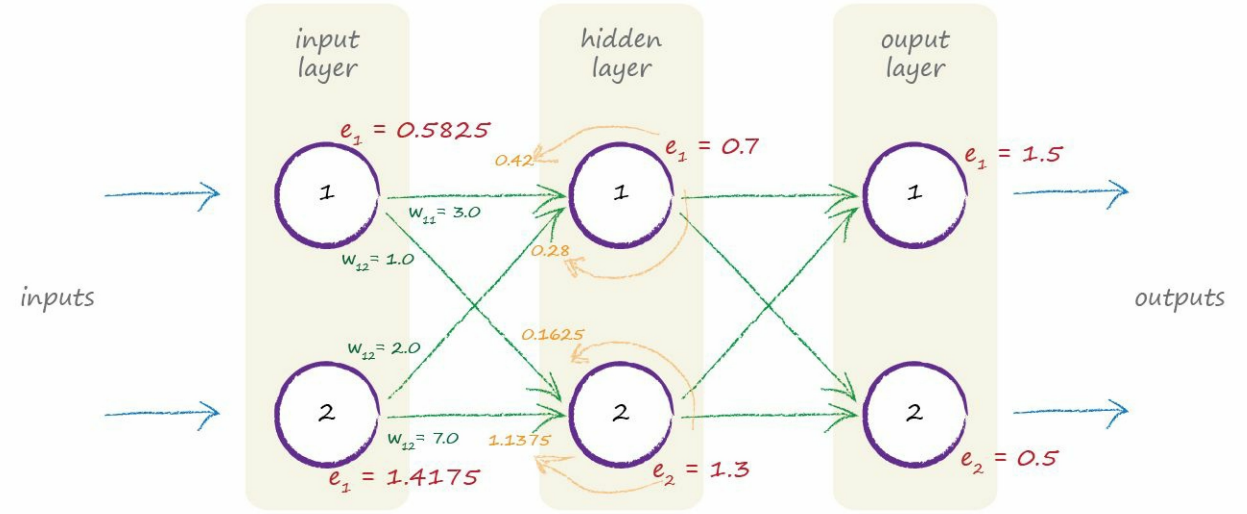
$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

It helps to see all this theory in action, so the following illustrates the propagation of errors back into a simple 3 layer network with actual numbers.



Let's follow one error back. You can see the error 0.5 at the second output layer node being split proportionately into 0.1 and 0.4 across the two connected links which have weights 1.0 and 4.0. You can also see that the recombined error at the second hidden layer node is the sum of the connected split errors, which here are 0.9 and 0.4, to give 1.3.

The next diagram shows the same idea applied to the preceding layer, working further back.



Key Points:

- Neural networks learn by refining their link weights. This is guided by the **error** - the difference between the right answer given by the training data and their actual output.
- The error at the output nodes is simply the difference between the desired and actual output.
- However the error associated with internal nodes is not obvious. One popular approach is to split the output layer errors in **proportion** to the size of the connected link weights, and then recombine these bits at each internal node.

Backpropagating Errors with Matrix Multiplication

Can we use matrix multiplication to simplify all that laborious calculation? It helped earlier when we were doing loads of calculations to feed forward the input signals.

To see if error backpropagation can be made more concise using matrix multiplication, let's write out the steps using symbols. By the way, this is called trying to **vectorise** the process. Being able to express a lot of calculations in matrix form makes it more concise for us to write down, and also allows computers to do all that work much more efficiently because they take advantage of the repetitive similarities in the calculations that need to be done.

The starting point is the errors that emerge from the neural network at the final output layer. Here we only have two nodes in the output layer, so these are $\mathbf{e_1}$ and $\mathbf{e_2}$.

$$error_{output} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Next we want to construct the matrix for the hidden layer errors. That might sound hard so let's do it bit by bit. The first bit is the first node in the hidden layer. If you look at the diagrams above again, you can see that the first node's error has two paths contributing to it. They are (output error $\mathbf{e_1} * \mathbf{w_{11}}$) and (output error $\mathbf{e_2} * \mathbf{w_{12}}$). Now look at the second hidden layer node and we can again see two paths contributing to its error, ($\mathbf{e_1} * \mathbf{w_{21}}$) and ($\mathbf{e_2} * \mathbf{w_{22}}$). So we have the following matrix for the hidden layer.

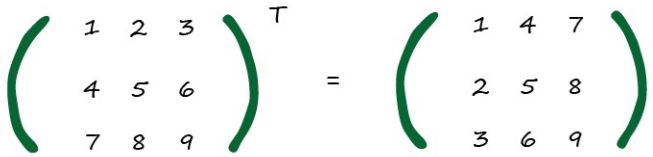
$$error_{hidden} = \begin{pmatrix} (e_1 * w_{11}) + (e_2 * w_{12}) \\ (e_1 * w_{21}) + (e_2 * w_{22}) \end{pmatrix}$$

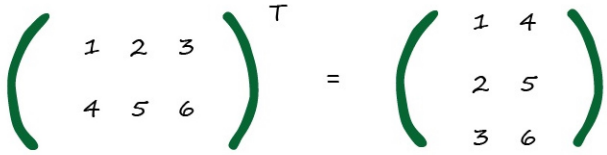
Some can spot the matrix multiplication easily, others will take a little longer staring at this expression! Here it is:

$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

That weight matrix is like the one we constructed before but has been flipped along a diagonal line so that the top right is now at the bottom left, and the bottom left is at the top right. This is called **transposing** a matrix, and is written as $\mathbf{w^T}$.

Here are two examples of a transposing matrix of numbers, so we can see clearly what happens. You can see this works even when the matrix has a different number of rows from columns.





So we have what we wanted, a matrix approach to propagating the errors back:

$$error_{hidden} = w^{T}_{hidden\_output} \cdot error_{output}$$

We've done a lot of work, a huge amount!

Key Points:

- Backpropagating the error can be expressed as a matrix multiplication.
- This allows us to express it concisely, irrespective of network size, and also allows computer languages that understand matrix calculations to do the work more efficiently and quickly.
- This means **both** feeding signals forward and error backpropagation can be made efficient using matrix calculations.

Take a well deserved break, because the next and final theory section is really very cool but does require a fresh brain.

How Do We Actually Update Weights?

We've 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 3 layers and 3 neurons in each layer, like 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 3 layer neural network with 3 nodes in each layer. The input at node **i** is $\mathbf{x_i}$, the weights for links connecting input node i to hidden node **j** is $\mathbf{w_{i,j}}$, similarly the output of hidden node **j** is $\mathbf{x_j}$, and the weights for links connecting hidden node **j** to output node **k** is $\mathbf{w_{j,k}}$. That funny symbol $\sum_a^b$ means sum the subsequent expression for all values between **a** and **b**.

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

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 3 layer neural network with 3 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!

You can see that the brute force approach isn't practical at all. In fact it gets worse very quickly as we add network layers, nodes or possibilities for weight values.

This puzzle resisted mathematicians for years, and was only really solved in a practical way as late as the 1960s-70s. There are different views on who did it first or made the key breakthrough but the important point is that this late discovery led to the explosion of modern neural networks which can carry out some very impressive tasks.

So how do we solve such an apparently hard problem? Believe it or not, you've already got the tools to do it yourself. We've covered all of them earlier. So let's get on with it.

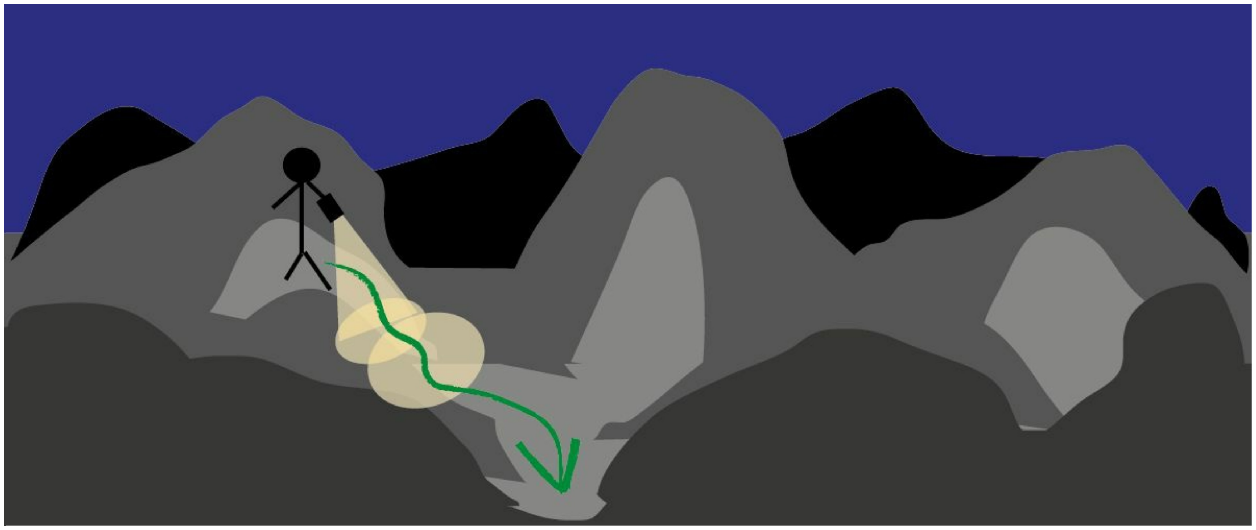
The first thing we must do is embrace **pessimism**.

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.

There are even more reasons to be pessimistic. 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. If we do that, we might find an approach which isn't mathematically perfect but does actually give us better results because it doesn't make false idealistic assumptions.

Let's illustrate what we mean by this. Imagine a very complex landscape with peaks and troughs, and hills with treacherous bumps and gaps. It's dark and you can't see anything. You know you're on the side of a hill and you need to get to the bottom. You don't have an accurate map of the entire landscape. You do have a torch. What do you do? You'll probably use the torch to look at the area close to your feet. You can't use it to see much further anyway, and certainly not the entire landscape. You can see which bit of earth seems to be going downhill and take small steps in that direction. In this way, you slowly work your way down the hill, step by step, without having a full map and without having worked out a journey beforehand.



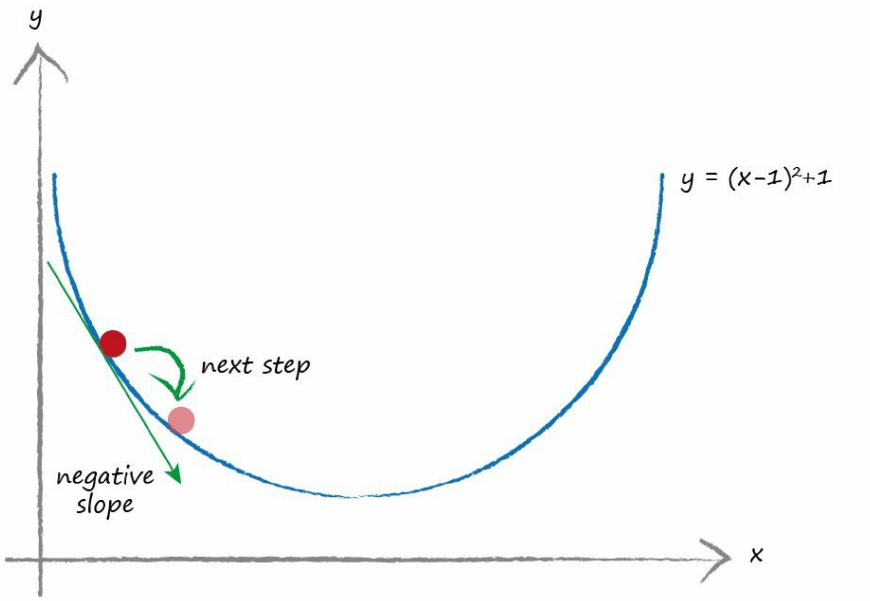
The mathematical version of this approach is called **gradient descent**, and you can see why. After you've taken a step, you look again at the surrounding area to see which direction takes you closer to your objective, and then you step again in that direction. You keep doing this until you're happy you've arrived at the bottom. The gradient refers to the slope of the ground. You step 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. If a function is so difficult that we can't easily find the minimum using algebra, we can use this method instead. Sure 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!

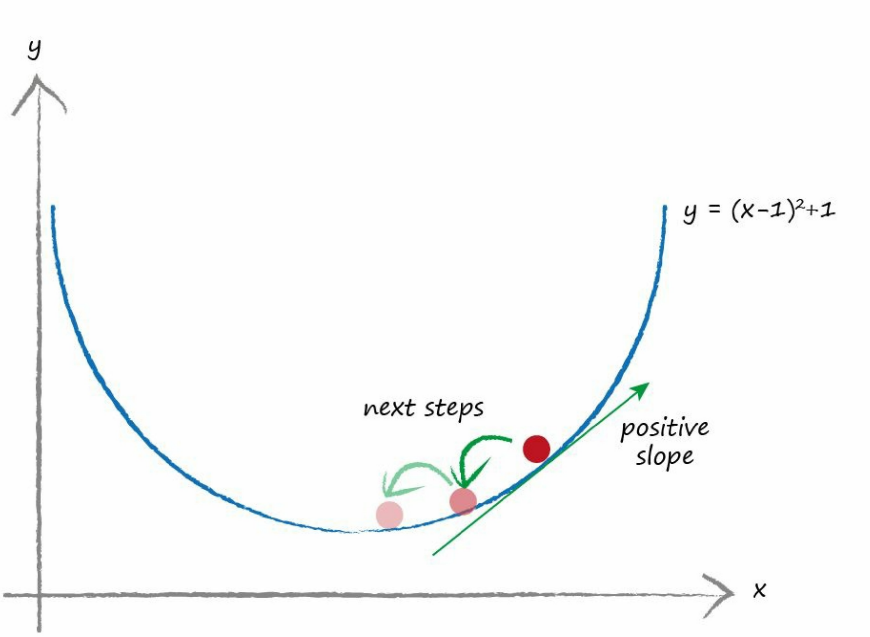
Let's look at this gradient descent idea with a super simple example so we can understand it properly.

The following graph shows a simple function $\mathbf{y} = (\mathbf{x}\text{-}1)^2 + 1$. If this was a function where y was the error, we would want to find the **x** which minimises it. For a moment, pretend this wasn't an easy function but instead a complex difficult one.



To do gradient descent we have to start somewhere. The graph shows our randomly chosen starting point. Like the hill climber, we look around the place we're standing and see which direction is downwards. The slope is marked on the graph and in this case is a negative gradient. We want to follow the downward direction so we move along x to the right. That is, we increase x a little. That's our hill climber's first step. You can see that we've improved our position and moved closer to the actual minimum.

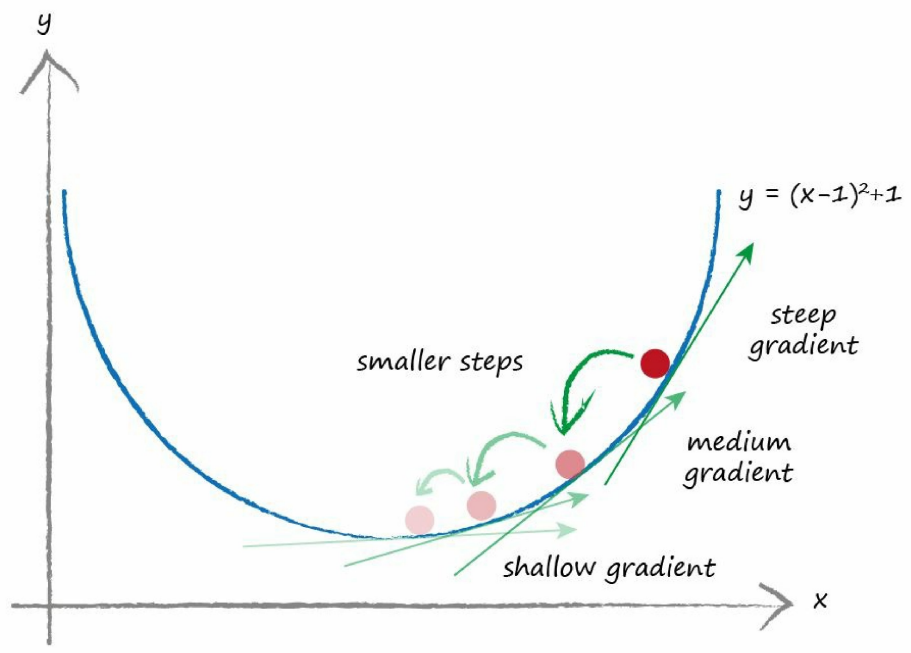
Let's imagine we started somewhere else, as shown in the next graph.



This time, the slope beneath our feet is positive, so we move to the left. That is, we decrease **x** a little. Again you can see we've improved our position by moving closer to the actual true minimum. We can keep doing this until our improvements are so small that we can be satisfied that we've arrived at the minimum.

A necessary refinement is to change the size of the steps we take to avoid overshooting the minimum and forever bouncing around it. You can imagine that if we've arrived 0.5 metres from the true minimum but can only take 2 metre steps then we're going to keep missing the minimum as every step we take in the direction of the minimum will overshoot. If we moderate the step size so it is proportionate to the size of the gradient then when we are close we'll take smaller steps. This assumes that as we get closer to a minimum the slope does indeed get shallower. That's not a bad assumption at all for most smooth continuous functions. It wouldn't be a good assumption for crazy zig-zaggy functions with jumps and gaps, which mathematicians call **discontinuities**.

The following illustrates this idea of moderating step size as the function gradient gets smaller, which is a good indicator of how close we are to a minimum.

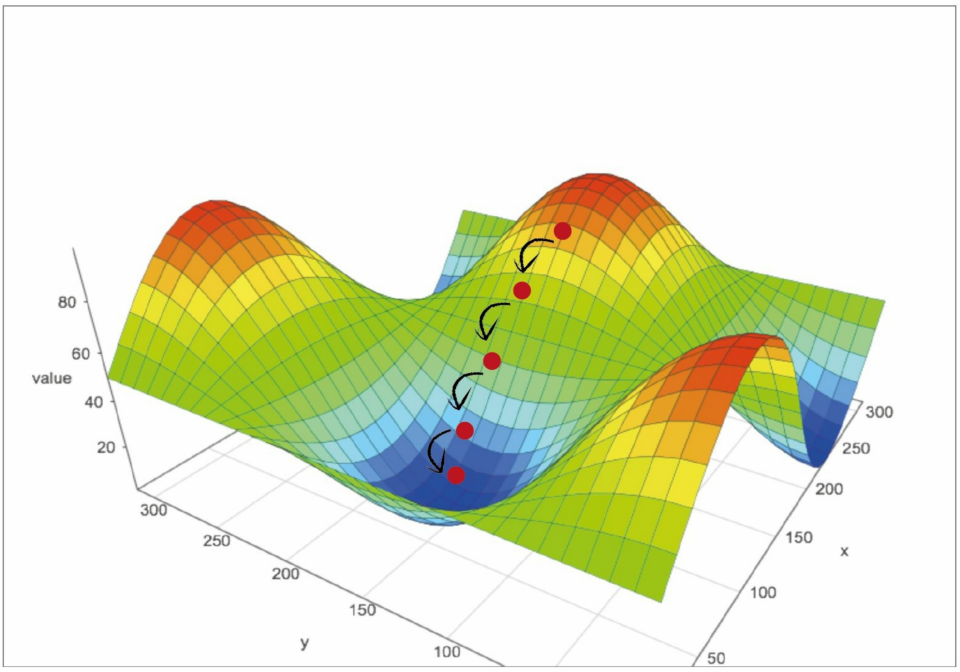


By the way, did you notice that we increase **x** in the opposite direction to the gradient? A positive gradient means we reduce **x**. A negative gradient means we increase **x**. The graphs make this clear, but it is easy to forget and get it the wrong way around.

When we did this gradient descent, we didn't work out the true minimum using algebra because we pretended the function $\mathbf{y} = (\mathbf{x}\text{-}1)^2 + 1$ was too complex and difficult. Even if we couldn't work out the slope exactly using mathematical precision, we could estimate it, and you can see this would still work quite well in moving us in the correct general direction.

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. Often hundreds of the them!

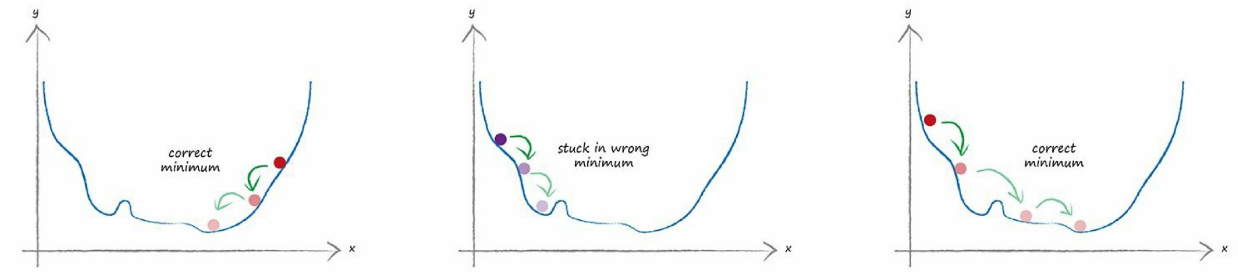
The following again illustrates gradient descent but with a slightly more complex function that depends on 2 parameters. This can be represented in 3 dimensions with the height representing the value of the function.



You may be looking at that 3-dimensional surface and wondering whether gradient descent ends up in that other valley also shown at the right. In fact, thinking more generally, doesn't gradient descent sometimes get stuck in the wrong valley, because some complex functions will have many valleys? What's the wrong valley? It's a valley which isn't the lowest. The answer to this is yes, that can happen.

To avoid ending up in the wrong valley, or function **minimum**, we train neural networks several times starting from different points on the hill to ensure we don't always ending up in the wrong valley. Different starting points means choosing different starting parameters, and in the case of neural networks this means choosing different starting link weights.

The following illustrates three different goes at gradient descent, with one of them ending up trapped in the wrong valley.



Let's pause and collect our thoughts.

Key Points:

- **Gradient descent** is a really good way of working out the minimum of a function, and it really works well when that function is so complex and difficult that we couldn't easily work it out mathematically using algebra.

- What's more, the method still works well when there are many parameters, something that causes other methods to fail or become impractical.

- This method is also **resilient** to imperfections in the data, we don't go wildly wrong if the function isn't quite perfectly described or we accidentally take a wrong step occasionally.

The output of a neural network is a complex difficult function with many parameters, the link weights, which influence its output. So we can use gradient descent to work out the right weights? Yes, as long as we pick the right error function.

The output function of a neural network itself isn't an error function. But we know we can turn it into one easily, because the error is the difference between the target training values and the actual output values.

There's something to watch out for here. Look at the following table of training and actual values for three output nodes, together with candidates for an error function.

Network Output	Target Output	Error (target - actual)	Error |target - actual|	Error $(target - actual)^2$
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

The first candidate for an error function is simply the **(target - actual)**. That seems reasonable enough, right? Well if you look at the sum over the nodes to get an overall figure for how well the network is trained, you'll see the sum is zero!

What happened? Clearly the network isn't perfectly trained because the first two node outputs are different to the target values. The sum of zero suggests there is no error. This happens because the positive and negative errors cancel each other out. Even if they didn't cancel out completely, you can see this is a bad measure of error.

Let's correct this by taking the **absolute** value of the difference. That means ignoring the sign, and is written **|target - actual|**. That could work, because nothing can ever cancel out. The reason this isn't popular is because the slope isn't continuous near the minimum and this makes gradient descent not work so well, because we can bounce around the V-shaped valley that this error function has. The slope doesn't get smaller closer to the minimum, so our steps don't get smaller, which means they risk overshooting.

The third option is to take the square of the difference $\mathbf{(target - actual)^2}$. There are several reasons why we prefer this third one over the second one, including the following:

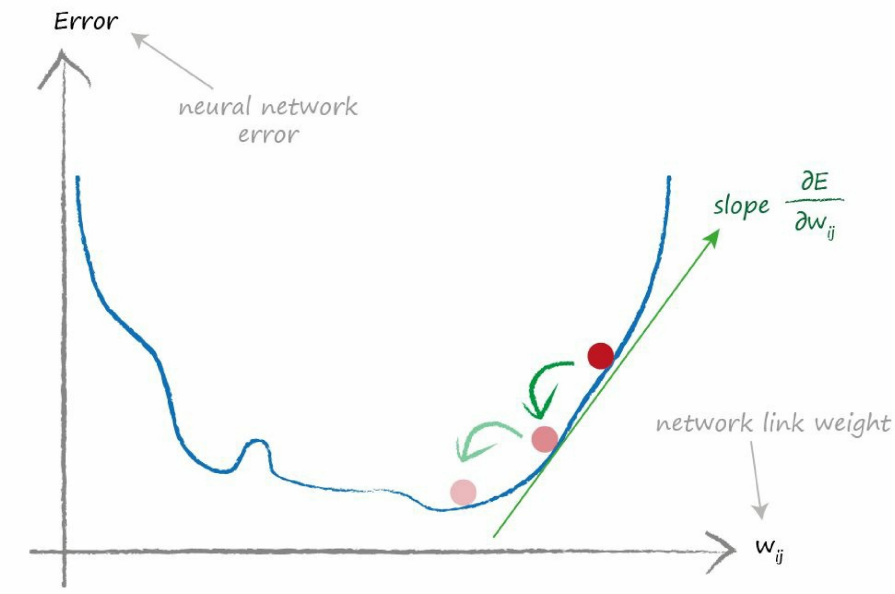
- The algebra needed to work out the slope for gradient descent is easy enough with this squared error.
- The error function is smooth and continuous making gradient descent work well - there are no gaps or abrupt jumps.
- The gradient gets smaller nearer the minimum, meaning the risk of overshooting the objective gets smaller if we use it to moderate the step sizes.

Is there 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.

Right, we're on the final lap now!

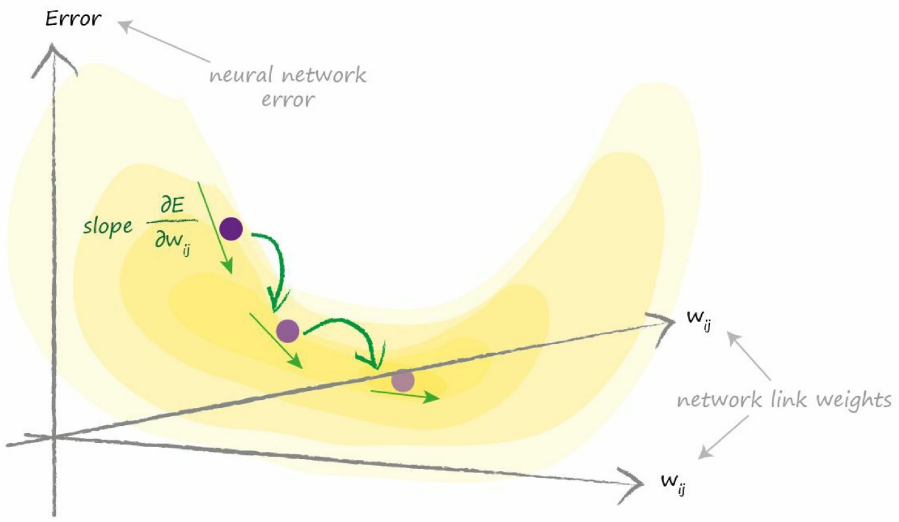
To do gradient descent, we now need to work out the slope of the error function with respect to the weights. This requires **calculus**. You may already be familiar with calculus, but if you're not, or just need a reminder, the **Appendix** contains a gentle introduction. Calculus is simply a mathematically precise way of working out how something changes when something else does. For example, how the length of a 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're trying to achieve.



The graph is just like the one we saw before to emphasise that we're not doing anything different. This time the function we're trying to minimise 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 minimise the error which is now more like a mountainous landscape with a valley.



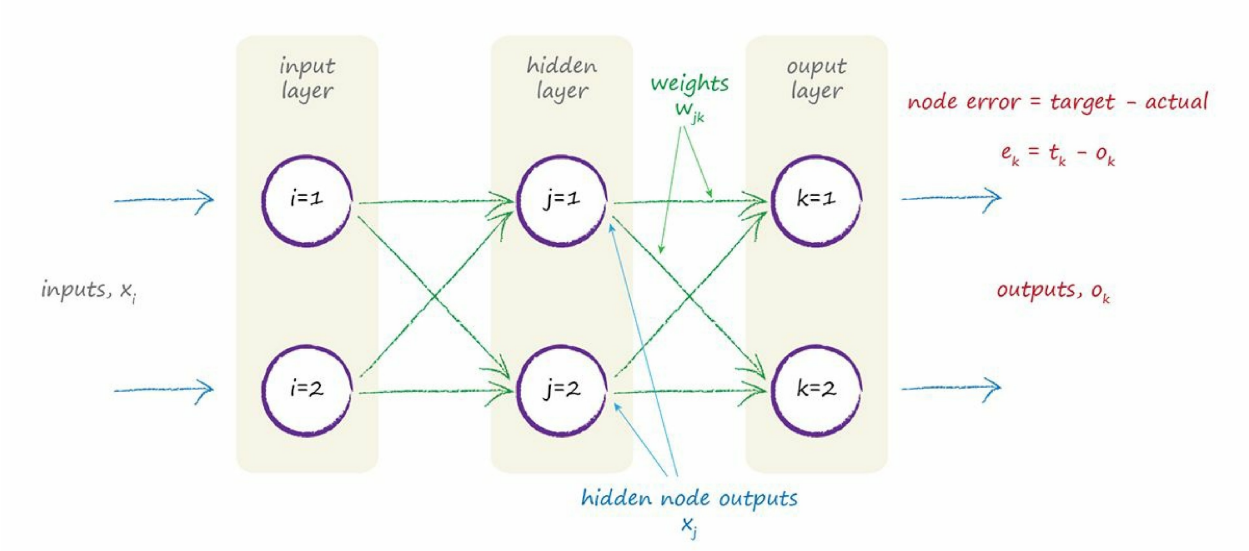
It's harder to visualise that error surface as a function of many more parameters, but the idea to use gradient descent to find the minimum is still the same.

Let's write out mathematically what we want.

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

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'll 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.

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

All we've 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 $\mathbf{o_n}$, only depends on the links that connect to it. That means for a node **k**, the output $\mathbf{o_k}$ only depends on weights $\mathbf{w_{jk}}$, 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 $\mathbf{w_{jb}}$, where **b** does not equal **k**, because there is no link connecting them. The weight $\mathbf{w_{jb}}$ is for a link connecting to output node **b** not **k**.

This means we can remove all the $\mathbf{o_n}$ from that sum except the one that the weight $\mathbf{w_{jk}}$ links to, that is $\mathbf{o_k}$. This removes that pesky sum totally! A nice trick worth keeping in your back pocket.

If you've had your coffee, you may have realised this means the error function didn't need to sum over all the output nodes in the first place. We've seen the reason is that the output of a node only depends on the connected links and hence their weights. This is often glossed over in many texts which simply state the error function without explaining it.

Anyway, we have a simpler expression.

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

Now, we'll do a bit of calculus. Remember, you can refer to the Appendix if you're unfamiliar with differentiation.

That $\mathbf{t_k}$ part is a constant, and so doesn't vary as $\mathbf{w_{jk}}$ varies. That is $\mathbf{t_k}$ isn't a function of $\mathbf{w_{jk}}$. If you think about it, it would be really strange of the truth examples which provide the target values did change depending on the weights! That leaves the $\mathbf{o_k}$ part which we know does depend on $\mathbf{w_{jk}}$ because the weights are used to feed forward the signal to become the outputs $\mathbf{o_k}$.

We'll use the chain rule to break apart this differentiation task into more manageable pieces. Again refer to the Appendix for an introduction to the chain rule.

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

Now we can attack each simpler bit in turn. The first bit is easy as we're taking a simple derivative of a squared function. This gives us the following.

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

The second bit needs a bit more thought but not too much. That $\mathbf{o_k}$ 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 $\mathbf{o_j}$ is the output from the previous hidden layer node, not the output from the final layer $\mathbf{o_k}$.

How do we differentiate the sigmoid function? We could do it the long hard way, using the fundamental ideas in the Appendix, but others have already done that work. We can just use the well known answer, just like mathematicians all over the world do every day.

$$\frac{\partial}{\partial x} sigmoid (x) = sigmoid (x) \left(1 - sigmoid (x) \right)$$

Some functions turn into horrible expressions when you differentiate them. This sigmoid has a nice simple and easy to use result. It's one of the reasons the sigmoid is popular for activation functions in neural networks.

So let's apply this cool result to get the following.

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

$$= -2(t_k - o_k) \cdot sigmoid (\Sigma_j w_{jk} \cdot o_j) (1 - sigmoid (\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

What's that extra last bit? It's the chain rule again applied to the sigmoid derivative because the expression inside the sigmoid() function also needs to be differentiated with respect to $\mathbf{w_{jk}}$. That too is easy and the answer is simply $\mathbf{o_j}$.

Before we write down the final answer, let's get rid of that 2 at the front. We can do that because we're only interested in the direction of the slope of the error function so we can descend it. It doesn't matter if there is a constant factor of 2, 3 or even 100 in front of that expression, as long we're consistent about which one we stick to. So let's get rid of it to keep things simple.

Here's the final answer we've been working towards, the one that describes the slope of the error function so we can adjust the weight $\mathbf{w_{jk}}$.

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

Phew! We did it!

That's the magic expression we've been looking for. The key to training neural networks.

It's worth a second look, and the colour coding helps show each part. The first part is simply the (target - actual) error we know so well. The sum expression inside the sigmoids is simply the signal into the final layer node, we could have called it $\mathbf{i_k}$ to make is look simpler. It's just the signal into a node before the activation squashing function is applied. That last part is the output from the previous hidden layer node **j**. It is worth viewing these expressions in these terms because you get a feel for what physically is involved in that slope, and ultimately the refinement of the weights.

That's a fantastic result and we should be really pleased with ourselves. Many people find getting to this point really hard.

One almost final bit of work to do. That expression we slaved over is for refining the weights between the hidden and output layers. We now need to finish the job and find a similar error slope for the weights between the input and hidden layers.

We could do loads of algebra again but we don't have to. We simply use that physical interpretation we just did and rebuild an expression for the new set of weights we're interested in. So this time,

- 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 $\mathbf{e_j}$.
- 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 $\mathbf{i_j}$.
- The last part is now the output of the first layer of nodes $\mathbf{o_i}$, which happen to be the input signals.

This nifty way of avoiding lots of work, is simply taking advantage of the symmetry in the problem to construct a new expression. We say it's simple but it is a very powerful technique, wielded by some of the most big-brained mathematicians and scientists. You can certainly impress your mates with this!

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.

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

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.

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

The updated weight $\mathbf{w_{jk}}$ 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 , is a factor which moderates the strength of these changes to make sure we don't overshoot. It's often called a **learning rate**.

This expression applies to the weights between the input and hidden layers too, not just to those between the hidden and output layers. The difference will be the error gradient, for which we have the two expressions above.

Before we can leave this, we need to see what these calculations look like if we try to do them as matrix multiplications. To help us, we'll do what we did before, which is to write out what each element of the weight change matrix should be.

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

values from next layer

values from previous layer

I've left out the learning rate as that's just a constant and doesn't really change how we organise our matrix multiplication.

The matrix of weight changes contains values which will adjust the weight $\mathbf{w_{j,k}}$ linking node **j** in one layer with the node **k** in the next. You can see that first bit of the expression uses values from the next layer (node **k**), and the last bit uses values from the previous layer (node **j**).

You might need to stare at the picture above for a while to see that the last part, the horizontal matrix with only a single row, is the transpose of the outputs from the previous layer $\mathbf{O_j}$. The colour coding shows that the dot product is the right way around. If you're not sure, try writing out the dot product with these the other way around and you'll see it doesn't work.

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.

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

That's it! Job done.

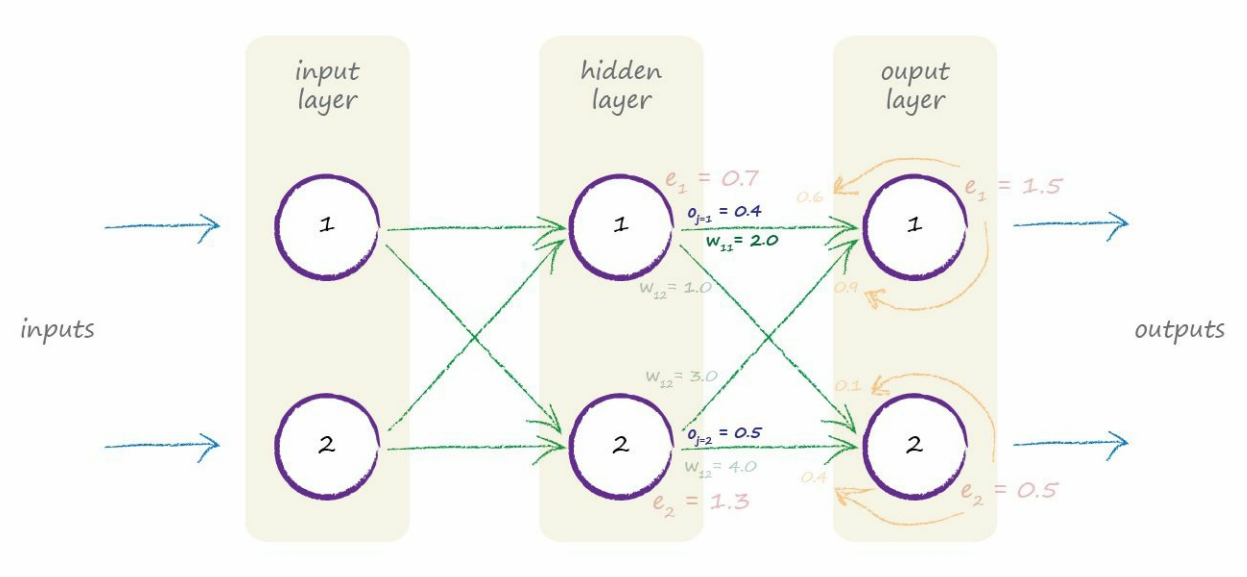
Key Points:

- 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.

Weight Update Worked Example

Let's work through a couple of examples with numbers, just to see this weight update method working.

The following network is the one we worked with before, but this time we've added example output values from the first hidden node $\mathbf{o_{j=1}}$ and the second hidden node $\mathbf{o_{j=2}}$. These are just made up numbers to illustrate the method and aren't worked out properly by feeding forward signals from the input layer.



We want to update the weight $\mathbf{w_{11}}$ between the hidden and output layers, which currently has the value 2.0.

Let's write out the error slope again.

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

Let's do this bit by bit:

- The first bit ($\mathbf{t_k - o_k}$) is the error $\mathbf{e_1}$ = 1.5, just as we saw before.
- The sum inside the sigmoid functions $\mathbf{\Sigma_j w_{jk} o_j}$ is (2.0 * 0.4) + (4.0 * 0.5) = 2.8.
- The sigmoid $1/(1 + e^{-2.8})$ is then 0.943. That middle expression is then 0.943 * (1 - 0.943) = 0.054.
- The last part is simply $\mathbf{o_j}$ which is $\mathbf{o_{j=1}}$ because we're interested in the weight $\mathbf{w_{11}}$ where $\mathbf{j}$ = 1. Here it is simply 0.4.

Multiplying all these three bits together and not forgetting the minus sign at the start gives us -0.06048.

If we have a learning rate of 0.1 that give is a change of - (0.1 * - 0.06048) = + 0.006. So the new $\mathbf{w_{11}}$ is the original 2.0 plus 0.006 = 2.006.

This is quite a small change, but over many hundreds or thousands of iterations the weights will eventually settle down to a configuration so that the well trained neural network produces outputs that reflect the training examples.

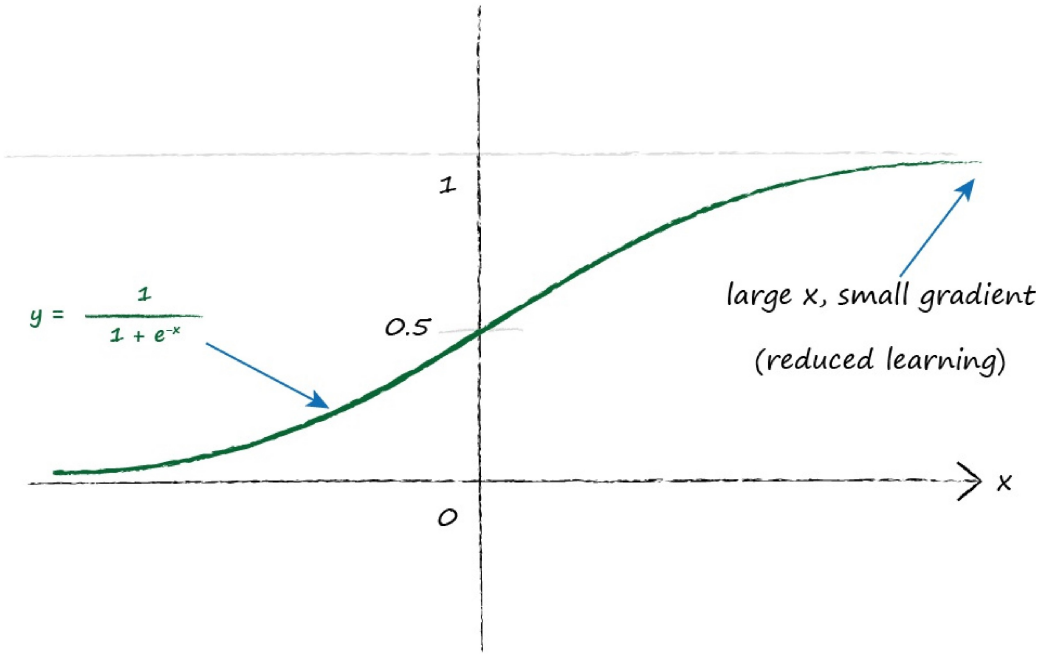
Preparing Data

In this section we're going to consider how we might best prepare the training data, prepare the initial random weights, and even design the outputs to give the training process a good chance of working.

Yes, you read that right! Not all attempts at using neural networks will work well, for many reasons. Some of those reasons can be addressed by thinking about the training data, the initial weights, and designing a good output scheme. Let's look at each in turn.

Inputs

Have a look at the diagram below of the sigmoid activation function. You can see that if the inputs are large, the activation function gets very flat.



A very flat activation function is problematic because we use the gradient to learn new weights. Look back at that expression for the weight changes. It depends on the gradient of the activation function. A tiny gradient means we've limited the ability to learn. This is called **saturating** a neural network. That means we should try to keep the inputs small.

Interestingly, that expression also depends on the incoming signal ($\mathbf{o_j}$) so we shouldn't make it too small either. Very very tiny values can be problematic too because computers can lose accuracy when dealing very very small or very very large numbers.

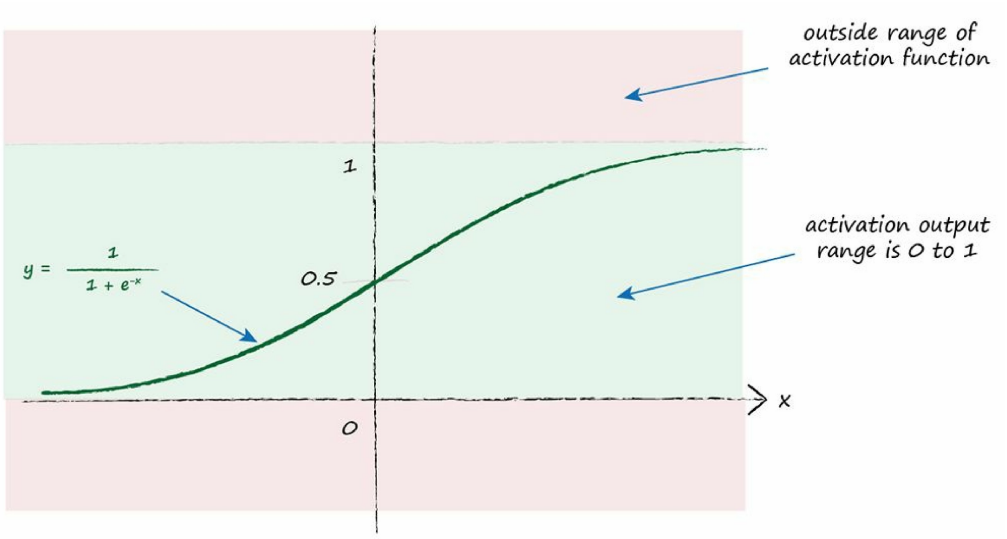
A good recommendation is to rescale inputs into the range 0.0 to 1.0. Some will add a small offset to the inputs, like 0.01, just to avoid having zero inputs which are troublesome because they kill the learning ability by zeroing the weight update expression by setting that $\mathbf{o_j} = 0$.

Outputs

The outputs of a neural network are the signals that pop out of the last layer of nodes. If we're using an activation function that can't produce a value above 1.0 then it would be silly to try to set larger

values as training targets. Remember that the logistic function doesn't even get to 1.0, it just gets ever closer to it. Mathematicians call this **asymptotically** approaching 1.0.

The following diagram makes clear that output values larger than 1.0 and below zero are simply not possible from the logistic activation function.



If we do set target values in these inaccessible forbidden ranges, the network training will drive ever larger weights in an attempt to produce larger and larger outputs which can never actually be produced by the activation function. We know that's bad as that saturates the network.

So we should rescale our target values to match the outputs possible from the activation function, taking care to avoid the values which are never really reached.

It is common to to use a range of 0.0 to 1.0, but some do use a range of 0.01 to 0.99 because both 0.0 and 1.0 are impossible targets and risk driving overly large weights.

Random Initial Weights

The same argument applies here as with the inputs and outputs. We should avoid large initial weights because they cause large signals into an activation function, leading to the saturation we just talked about, and the reduced ability to learn better weights.

We could choose initial weights randomly and uniformly from a range -1.0 to +1.0. That would be a much better idea than using a very large range, say -1000 to +1000.

Can we do better? Probably.

Mathematicians and computer scientists have done the maths to work out a rule of thumb for setting the random initial weights given specific shapes of networks and with specific activation functions. That's a lot of "specifics"! Let's carry on anyway.

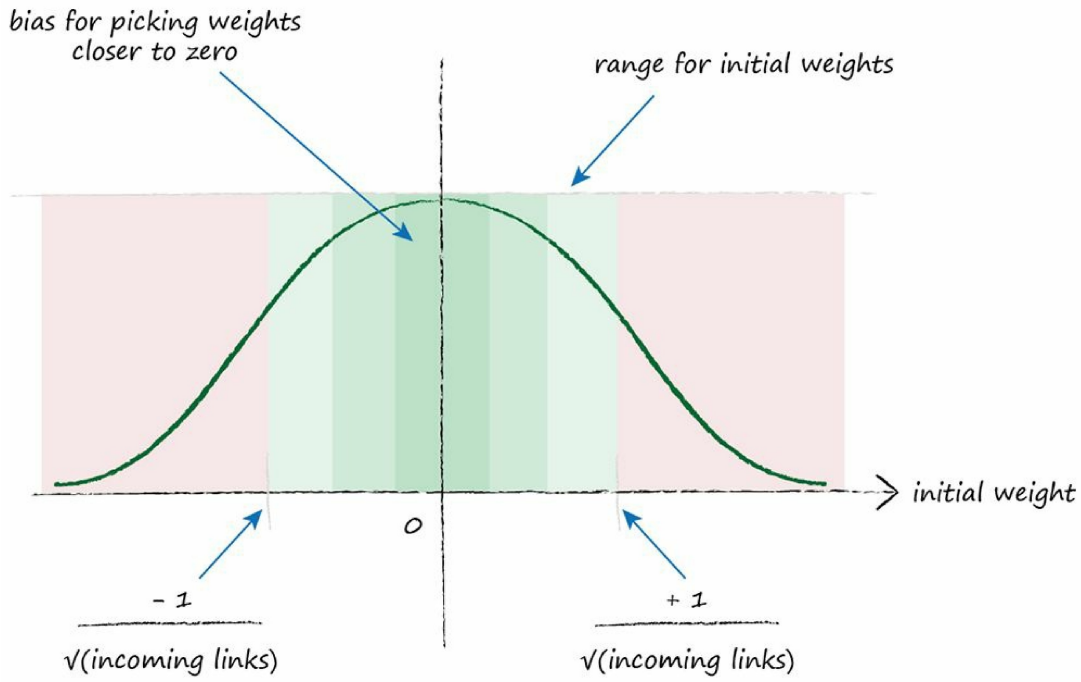
We won't go into the details of that working out but the core idea is that if we have many signals into a node, which we do in a neural network, and that these signals are already well behaved and not too large or crazily distributed, the weights should support keeping those signals well behaved as they are combined and the activation function applied. In other words, we don't want the weights to

undermine the effort we put into carefully scaling the input signals. The rule of thumb these mathematicians arrive at is that the weights are initialised randomly sampling from a range that is roughly the inverse of the square root of the number of links into a node. So if each node has 3 links into it, the initial weights should be in the range $1/(\sqrt{3}) = 0.577$. If each node has 100 incoming links, the weights should be in the range $1/(\sqrt{100}) = 0.1$.

Intuitively this make sense. Some overly large initial weights would bias the activation function in a biased direction, and very large weights would **saturate** the activation functions. And the more links we have into a node, the more signals are being added together. So a rule of thumb that reduces the weight range if there are more links makes sense.

If your are already familiar with the idea of sampling from probability distributions, this rule of thumb is actually about sampling from a normal distribution with mean zero and a standard deviation which is the inverse of the square root of the number of links into a node. But let's not worry too much about getting this precisely right because that rule of thumb assumes quite a few things which may not be true, such as an activation function like the alternative tanh() and a specific distribution of the input signals.

The following diagram summarises visually both the simple approach, and the more sophisticated approach with a normal distribution.



Whatever you do, don't set the initial weights the same constant value, especially no zero. That would be bad!

It would be bad because each node in the network would receive the same signal value, and the output out of each output node would be the same. If we then proceeded to update the weights in the network by back propagating the error, the error would have to be divided equally. You'll remember the error is split in proportion to the weights. That would lead to equal weight updates leading again to another set of equal valued weights. This symmetry is bad because if the properly trained network should have unequal weights (extremely likely for almost all problems) then you'd never get there.

Zero weights are even worse because they kill the input signal. The weight update function, which depends on the incoming signals, is zeroed. That kills the ability to update the weights completely.

The are many other things you can do to refine how you prepare your input data, how you set your weights, and how you organise your desired outputs. For this guide, the above ideas are both easy enough to understand and also have a decent effect, so we'll stop there.

Key Points:

- Neural networks don't work well if the input, output and initial weight data is not prepared to match the network design and the actual problem being solved.
- A common problem is **saturation** - where large signals, sometimes driven by large weights, lead to signals that are at the very shallow slopes of the activation function. This reduces the ability to learn better weights.
- Another problem is **zero** value signals or weights. These also kill the ability to learn better weights.
- The internal link weights should be **random** and **small**, avoiding zero. Some will use more sophisticated rules, for example, reducing the size of these weights if there are more links into a node.
- **Inputs** should be scaled to be small, but not zero. A common range is 0.01 to 0.99, or -1.0 to +1.0, depending on which better matches the problem.
- **Outputs** should be within the range of what the activation function can produce. Values below 0 or above 1, inclusive, are impossible for the logistic sigmoid. Setting training targets outside the valid range will drive ever larger weights, leading to saturation. A good range is 0.01 to 0.99.