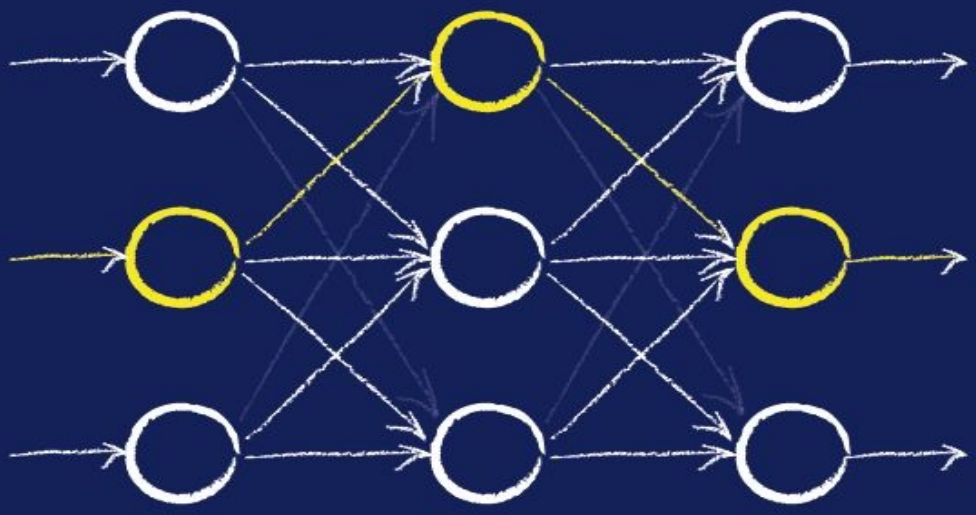
MAKE YOUR OWN NEURAL NETWORK



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.

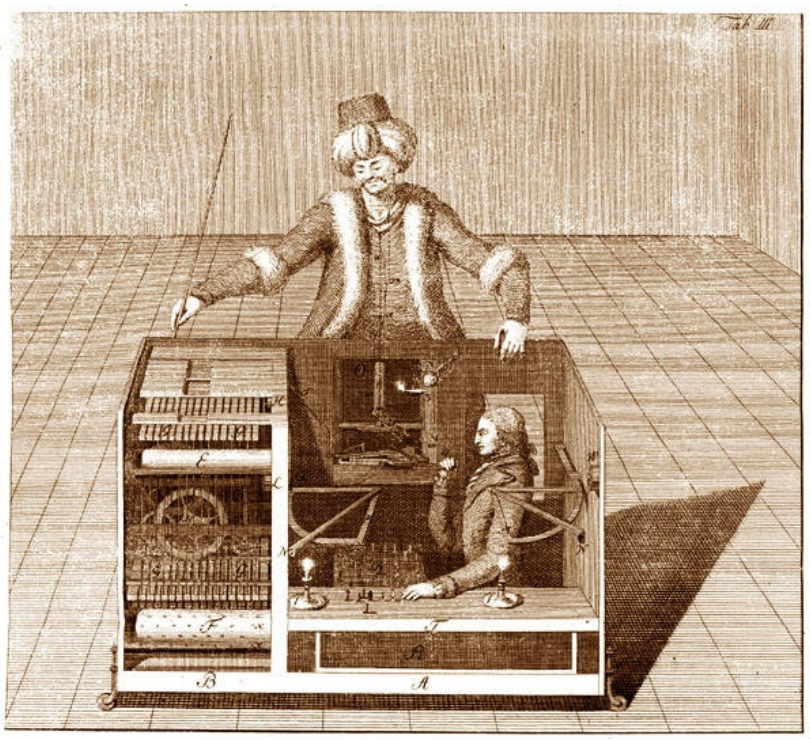
We've not been satisfied by mechanical or electronic machines helping us with simple tasks - flint sparking fires, pulleys lifting heavy rocks, and calculators doing arithmetic.

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.

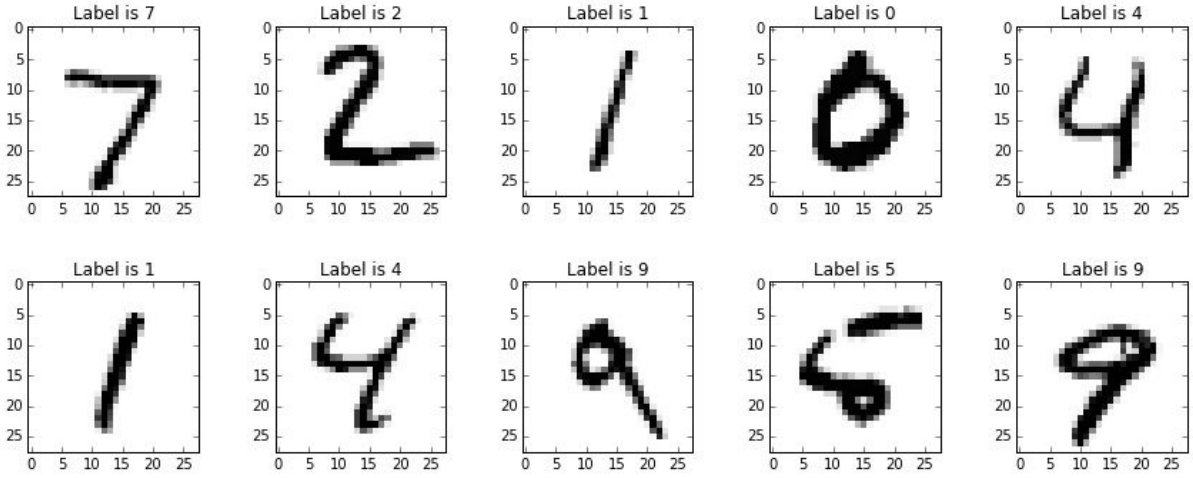
After a period of not much progress an incredibly powerful idea emerged to lift the search for machine intelligence out of its rut. Why not try to build artificial brains by copying how real biological brains worked? Real brains with neurons instead of logic gates, softer more organic reasoning instead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



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:

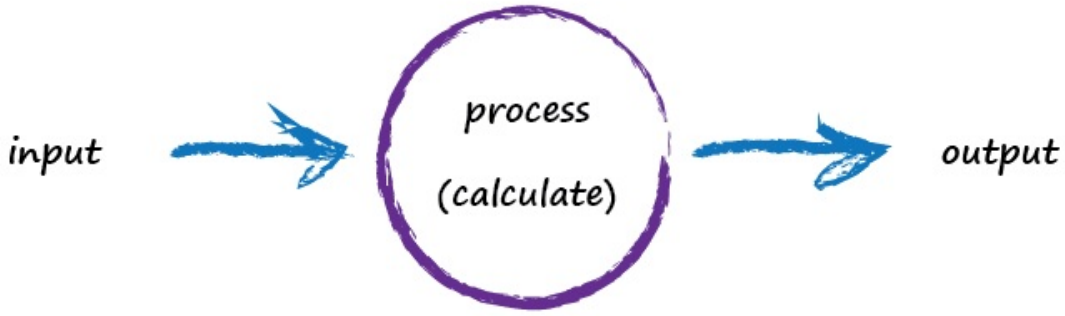
- Some tasks are easy for traditional computers, but hard for humans. For example, multiplying millions of pairs of numbers.
- On the other hand, some tasks are hard for traditional computers, but easy for humans. For example, recognising faces in a photo of a crowd.

A Simple Predicting Machine

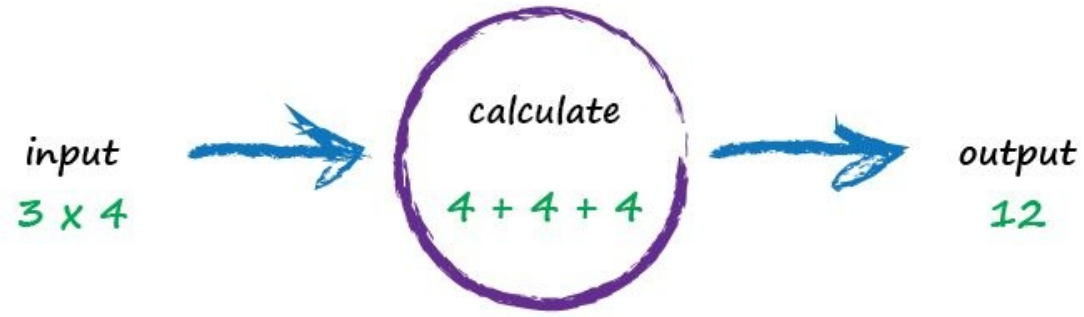
Let’s start super simple and build up from there.

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:



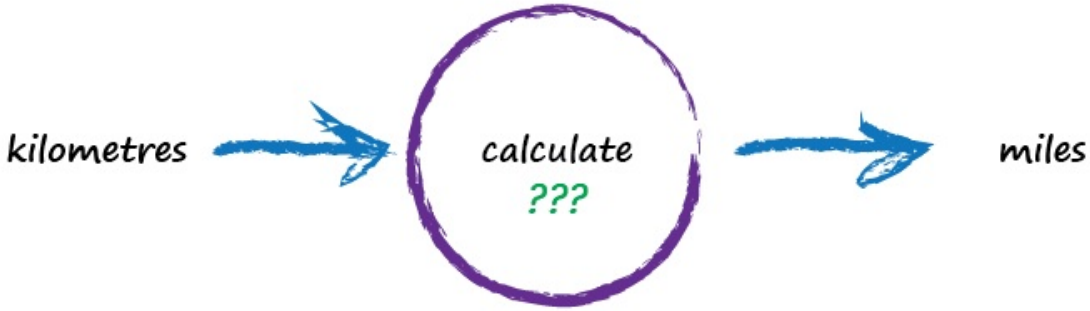
A computer takes some input, does some calculation and pops out an output. The following illustrates this. An input of “3 x 4” is processed, perhaps by turning multiplication into an easier set of additions, and the output answer “12” pops out.



“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:



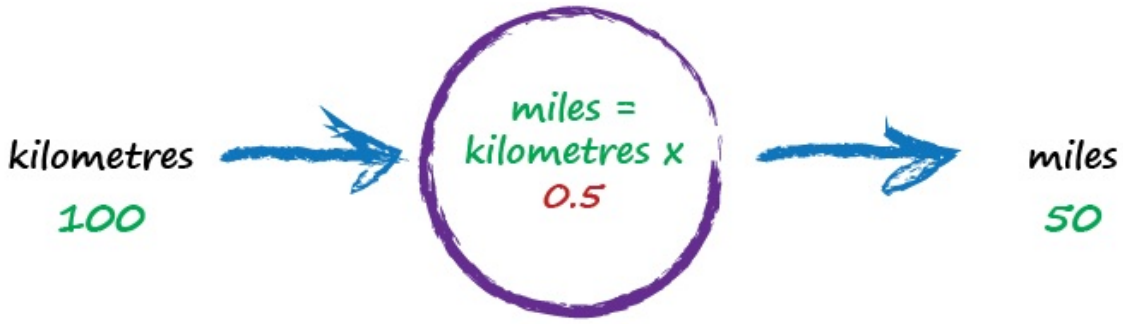
Now imagine we don’t know the formula for converting between kilometres and miles. All we know is the the relationship between the two is **linear**. That means if we double the number in miles, the same distance in kilometres is also doubled. That makes intuitive sense. The universe would be a strange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

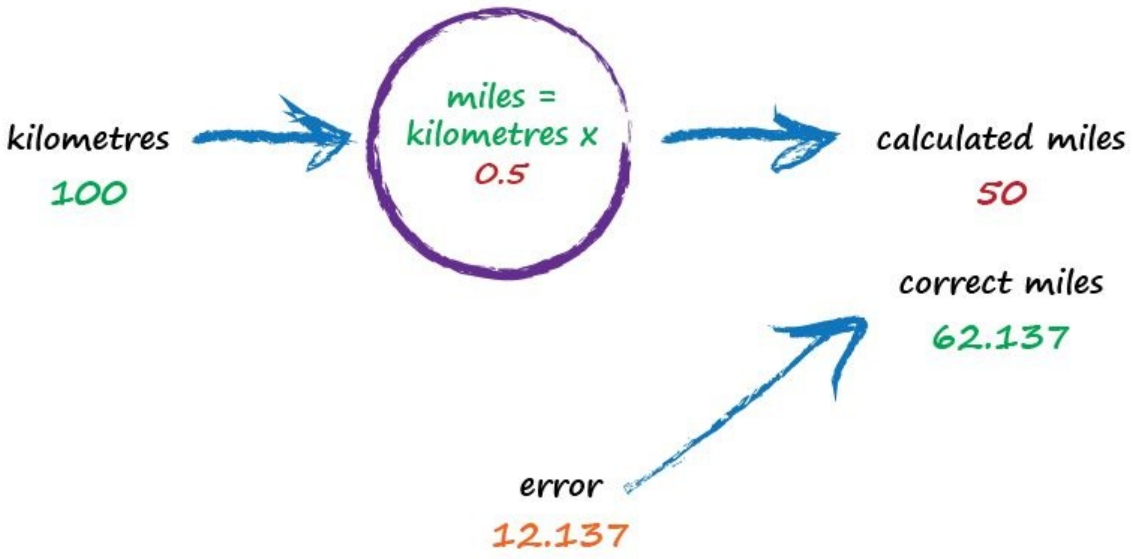


Here we have miles = kilometres x c, where kilometres is 100 and c is our current guess at 0.5. That gives 50 miles.

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,

error = truth - calculated
= 62.137 - 50
= 12.137



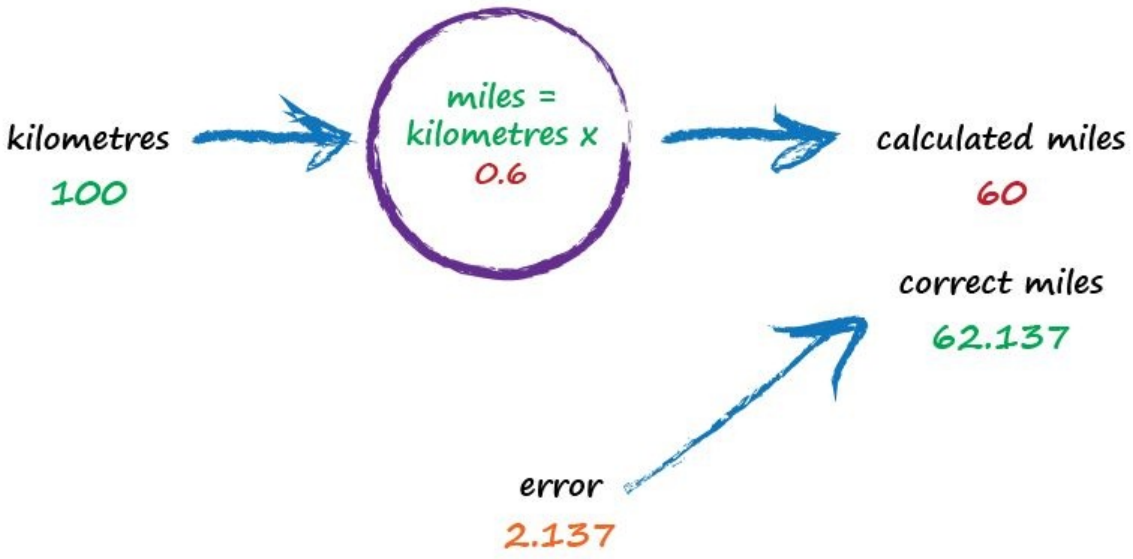
So what next? We know we're wrong, and by how much. Instead of being a reason to despair, we use this error to guide a second, better, guess at **c**.

Look at that error again. We were short by 12.137. 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!

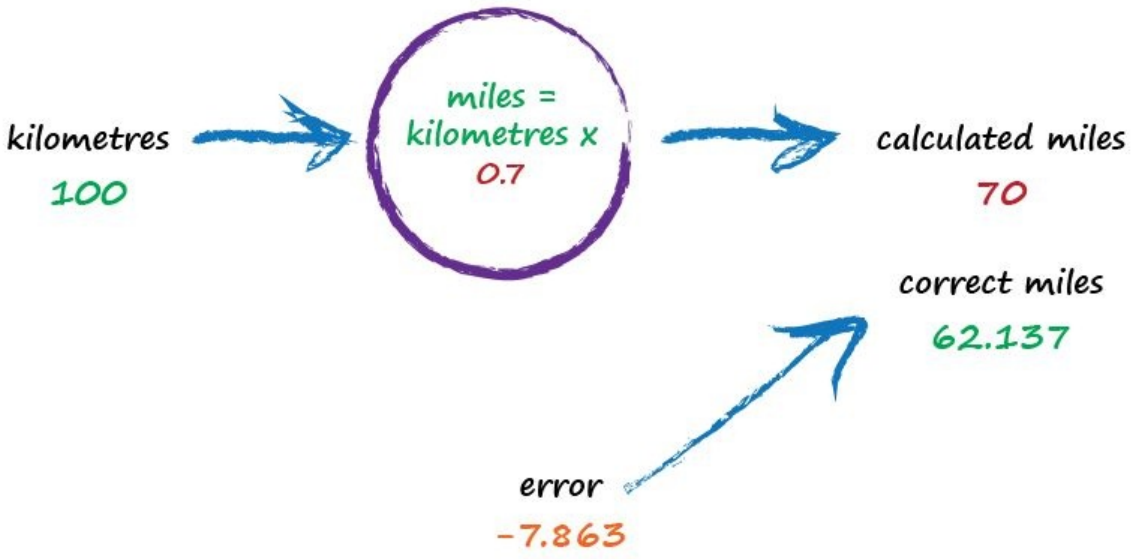
Now the error is a much smaller 2.137. It might even be an error we're happy to live with.



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.

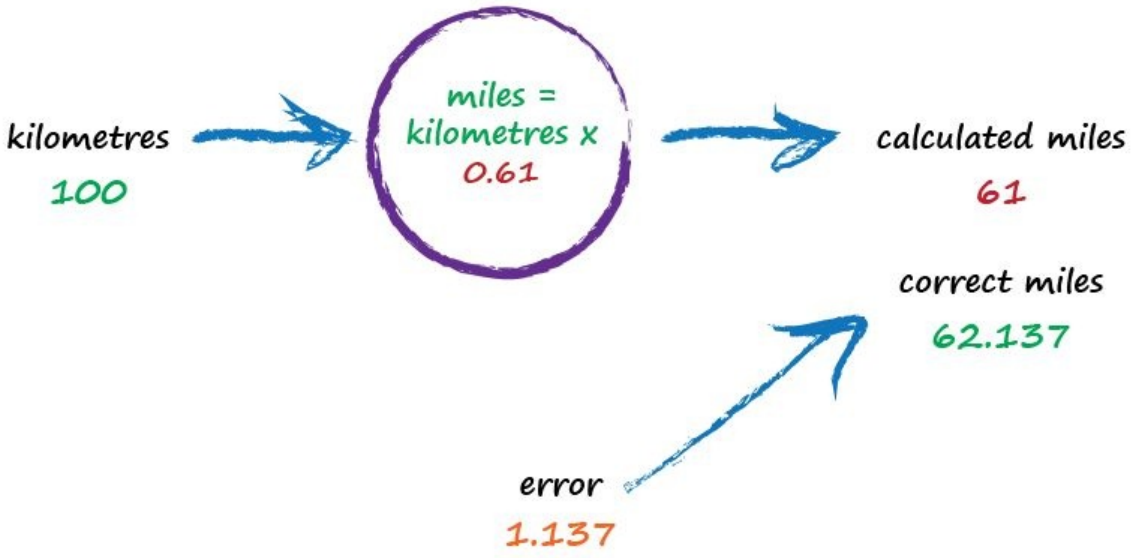
Rather than try to use algebra to work out the exact amount **c** needs to change, let's continue with this approach of refining c. If you're not convinced, and think it's easy enough to work out the exact answer, remember that many more interesting problems won't have simple mathematical formulae relating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

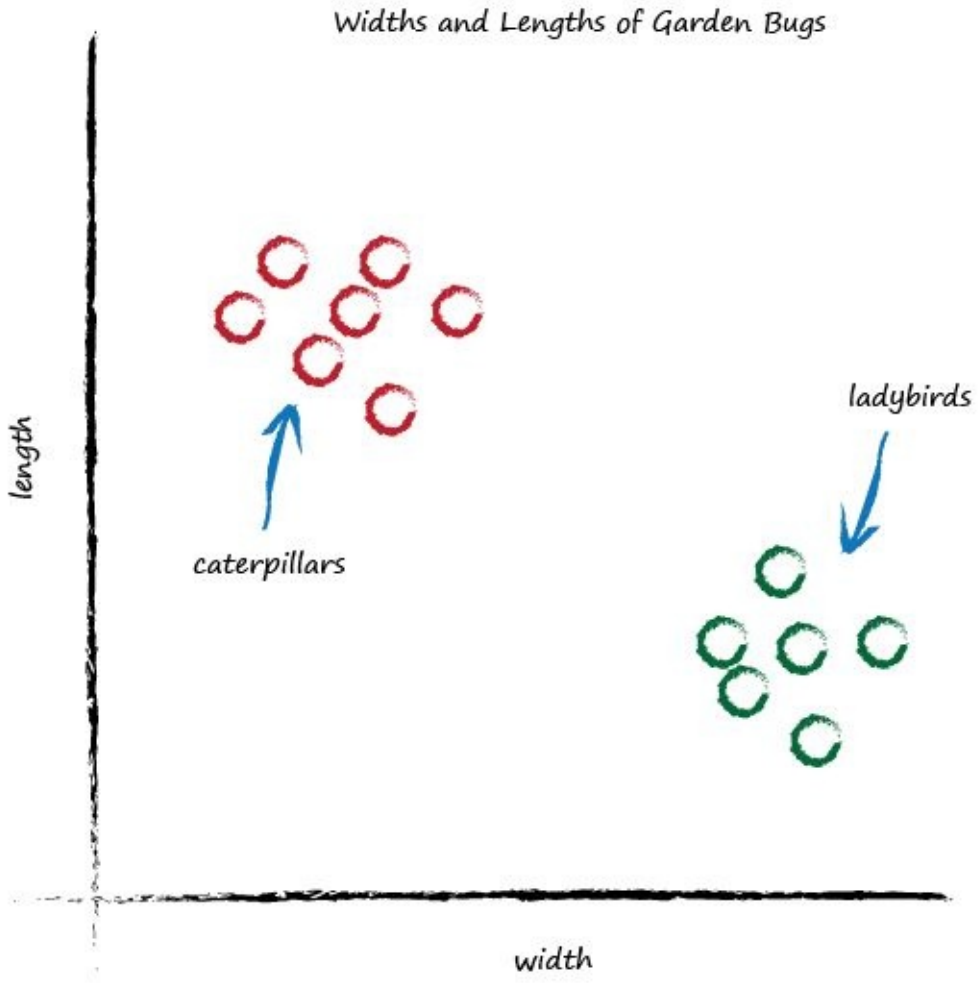
Key Points:

- 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.

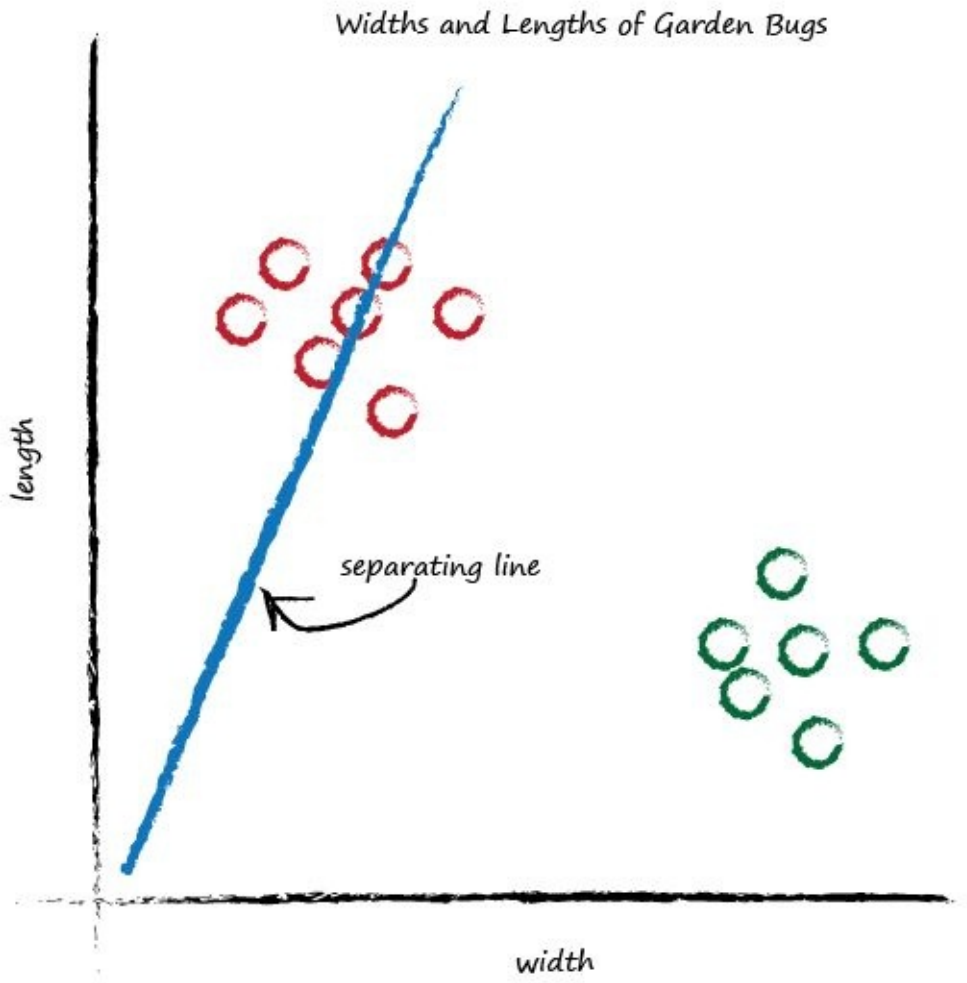
Now look at the following graph showing the measured widths and lengths of garden bugs.



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.

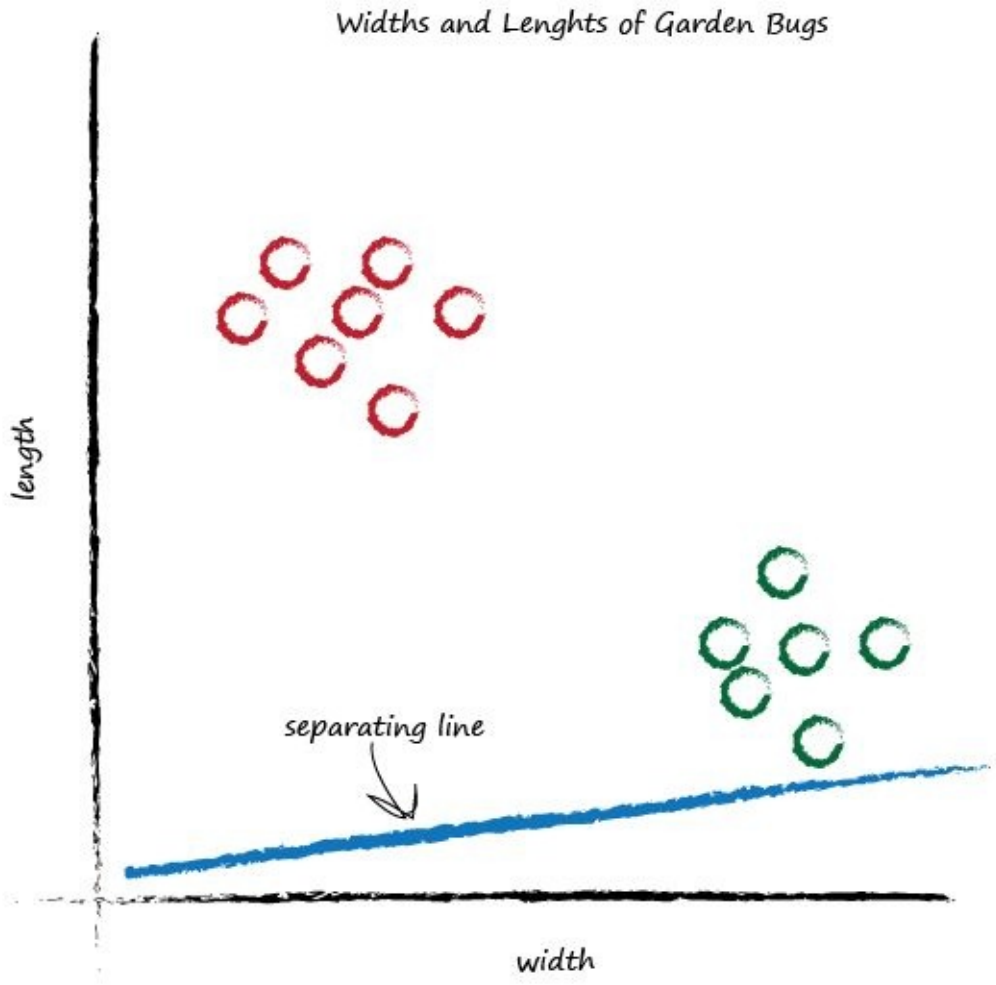
What happens if we place a straight line over that plot?



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.

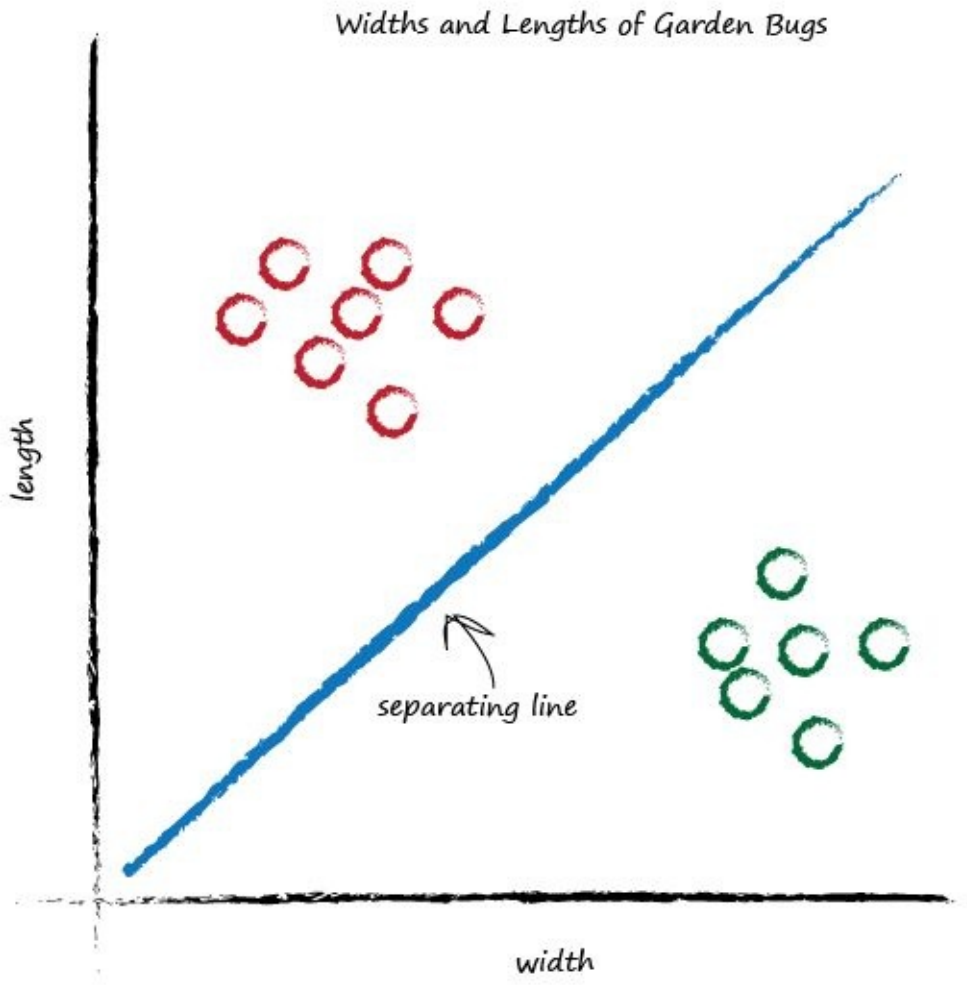
In the plot above, if the line was dividing the caterpillars from the ladybirds, then it could be used to **classify** an unknown bug based on its measurements. The line above doesn't do this yet because half the caterpillars are on the same side of the dividing line as the ladybirds.

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:

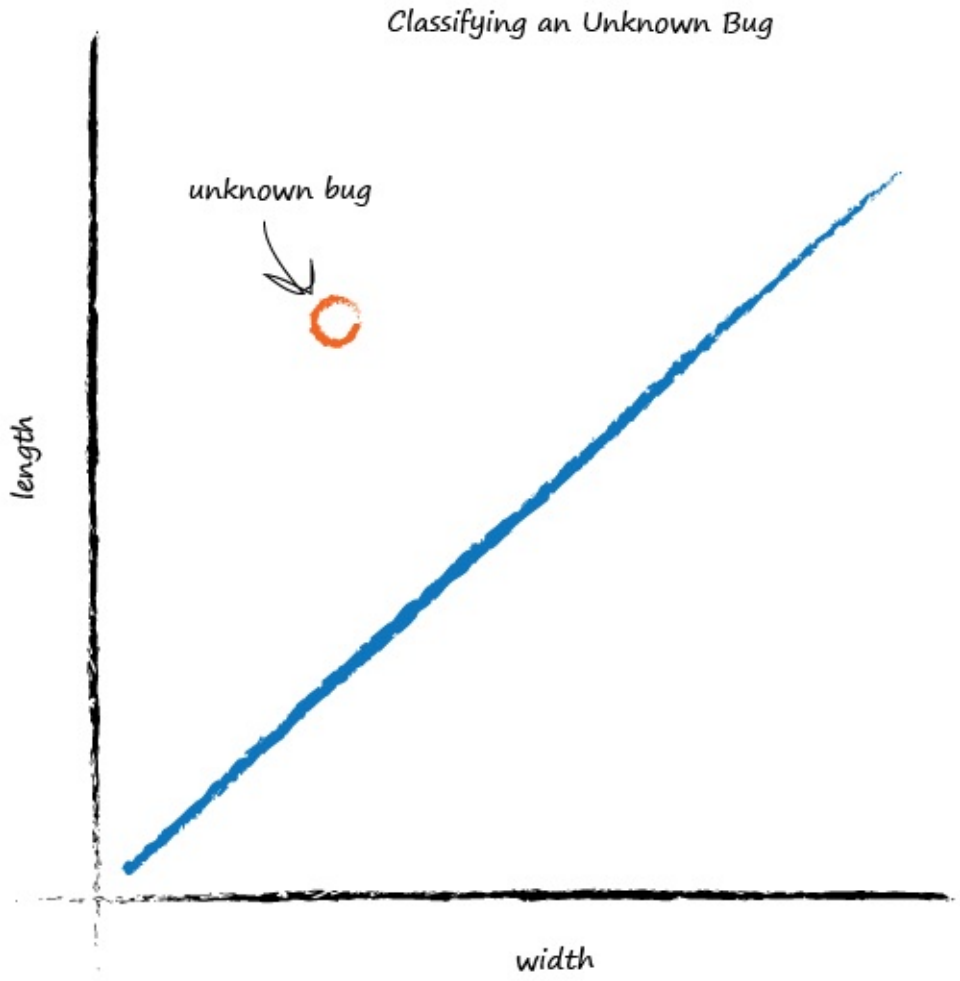


That’s much better! This line neatly separates caterpillars from ladybirds. We can now use this line as a **classifier** of bugs.

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.

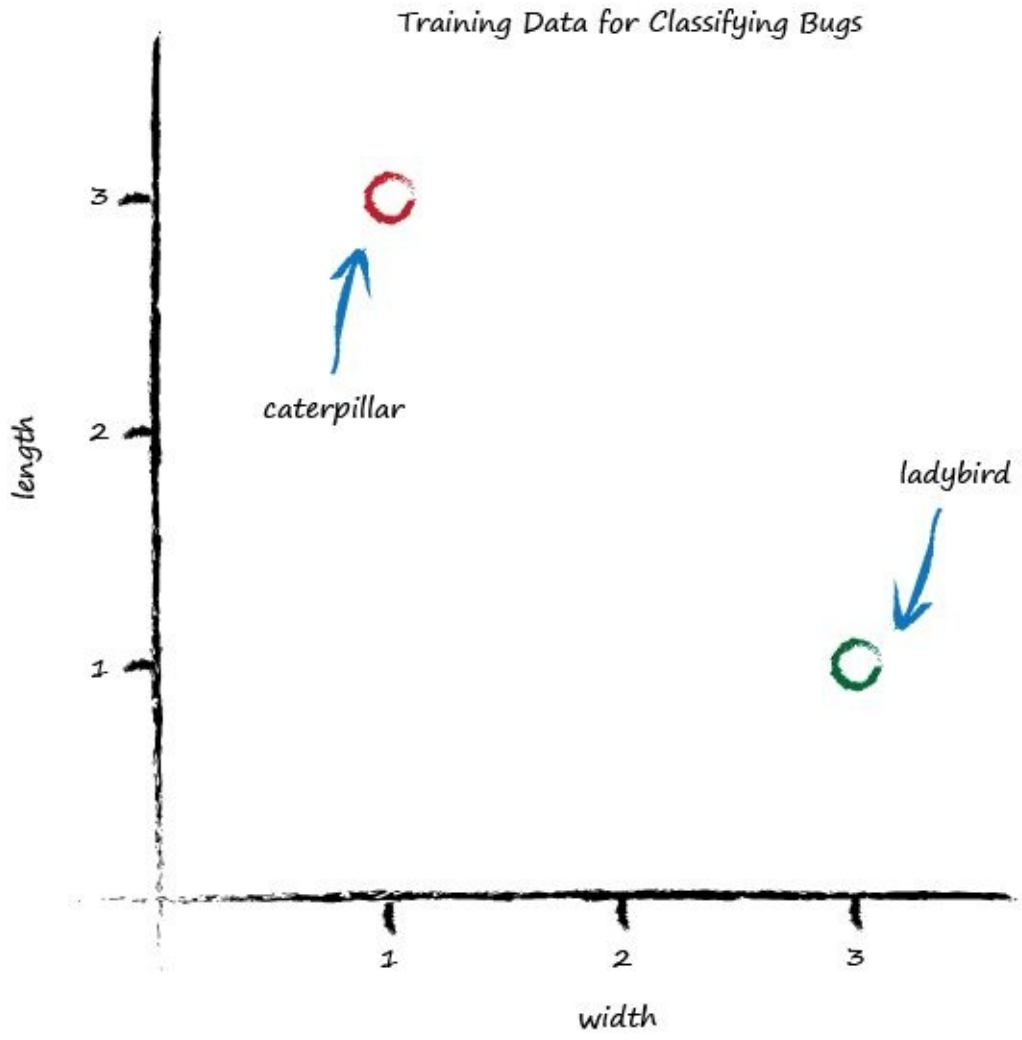
We do need some examples to learn from. The following table shows two examples, just to keep this exercise simple.

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:

$$\mathbf{y = Ax}$$

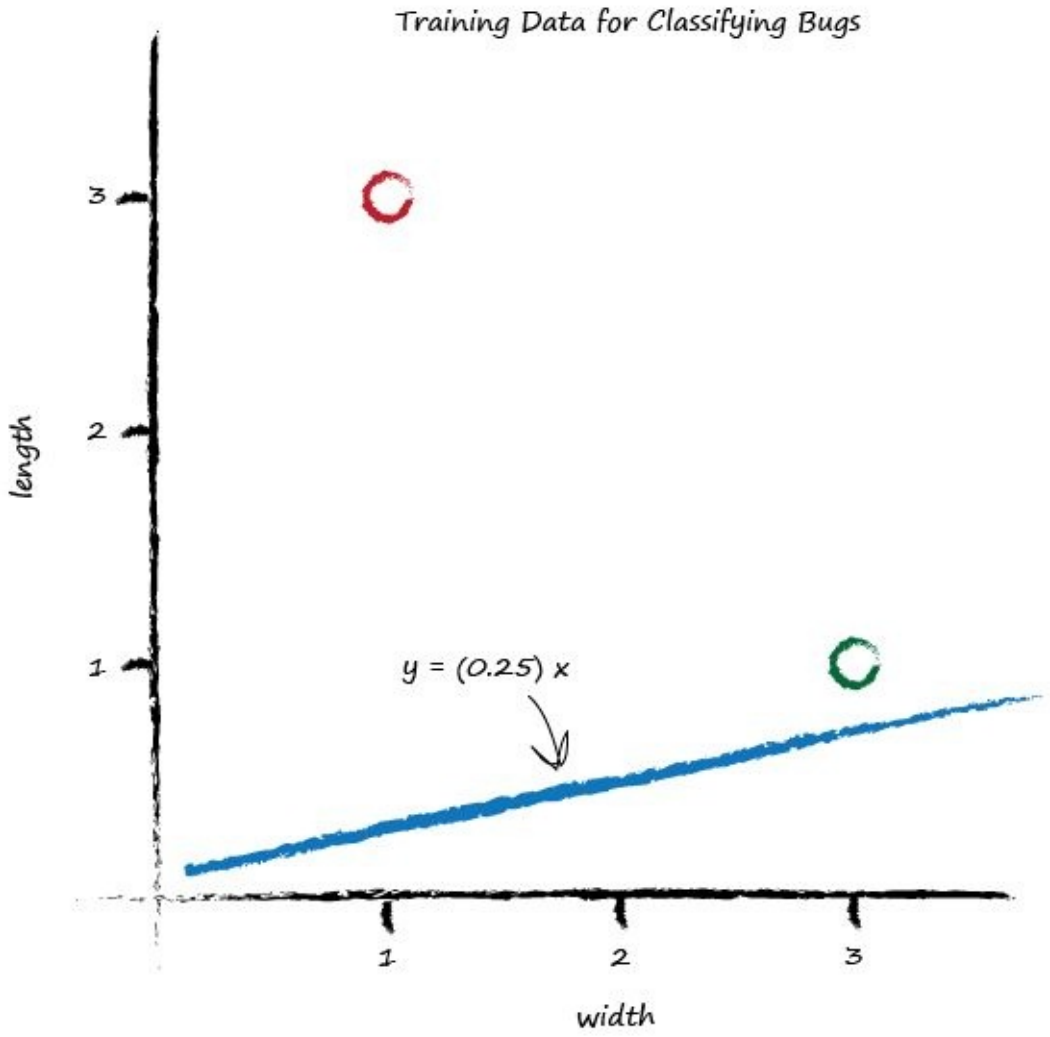
We’ve deliberately used the names **y** and **x** instead of length and width, because strictly speaking, the line is not a predictor here. It doesn’t convert width to length, like we previously converted miles to kilometres. Instead, it is a dividing line, a classifier.

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.

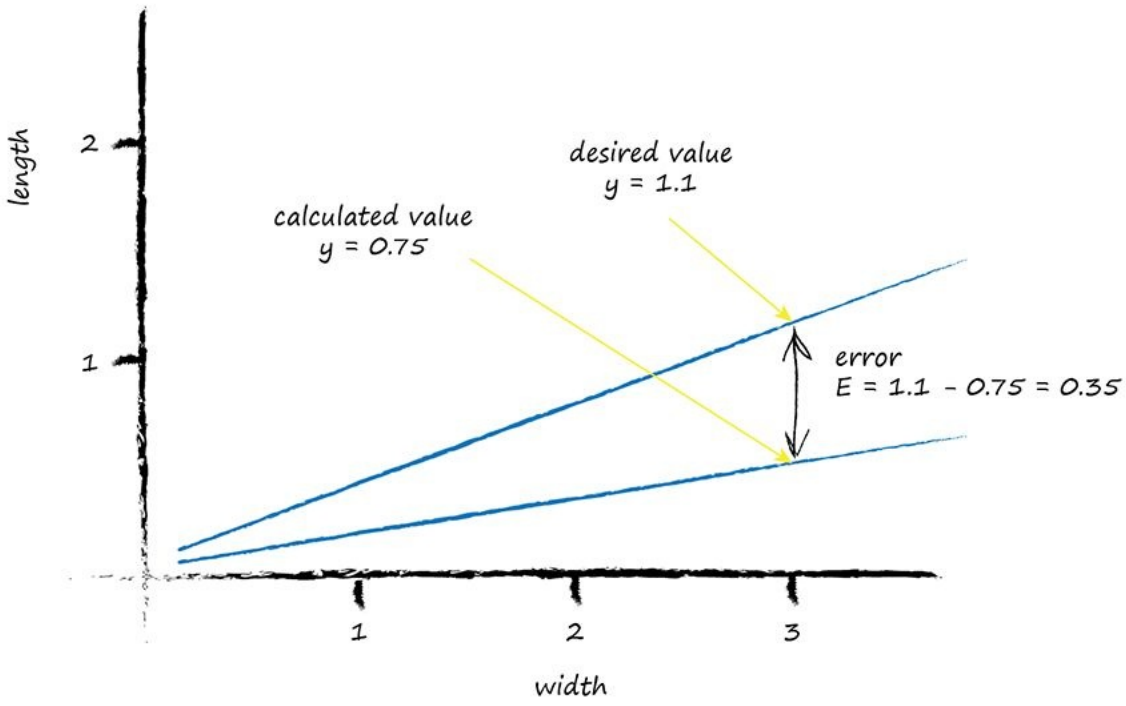
So the desired target is 1.1, and the error **E** is

$$\mathbf{error = (desired\ target - actual\ output)}$$

Which is,

$$\mathbf{E = 1.1 - 0.75 = 0.35}$$

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.

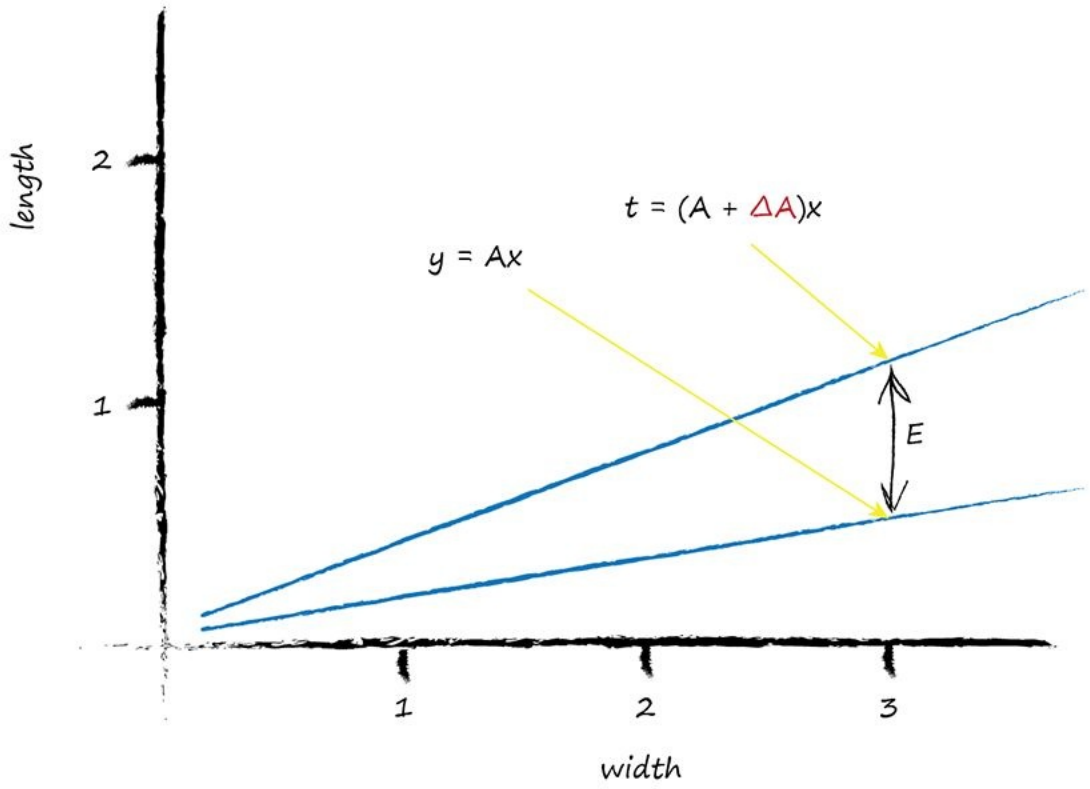
Let's start with the linear function for the classifier:

$$\mathbf{y = Ax}$$

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 **ΔA** 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:

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

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.

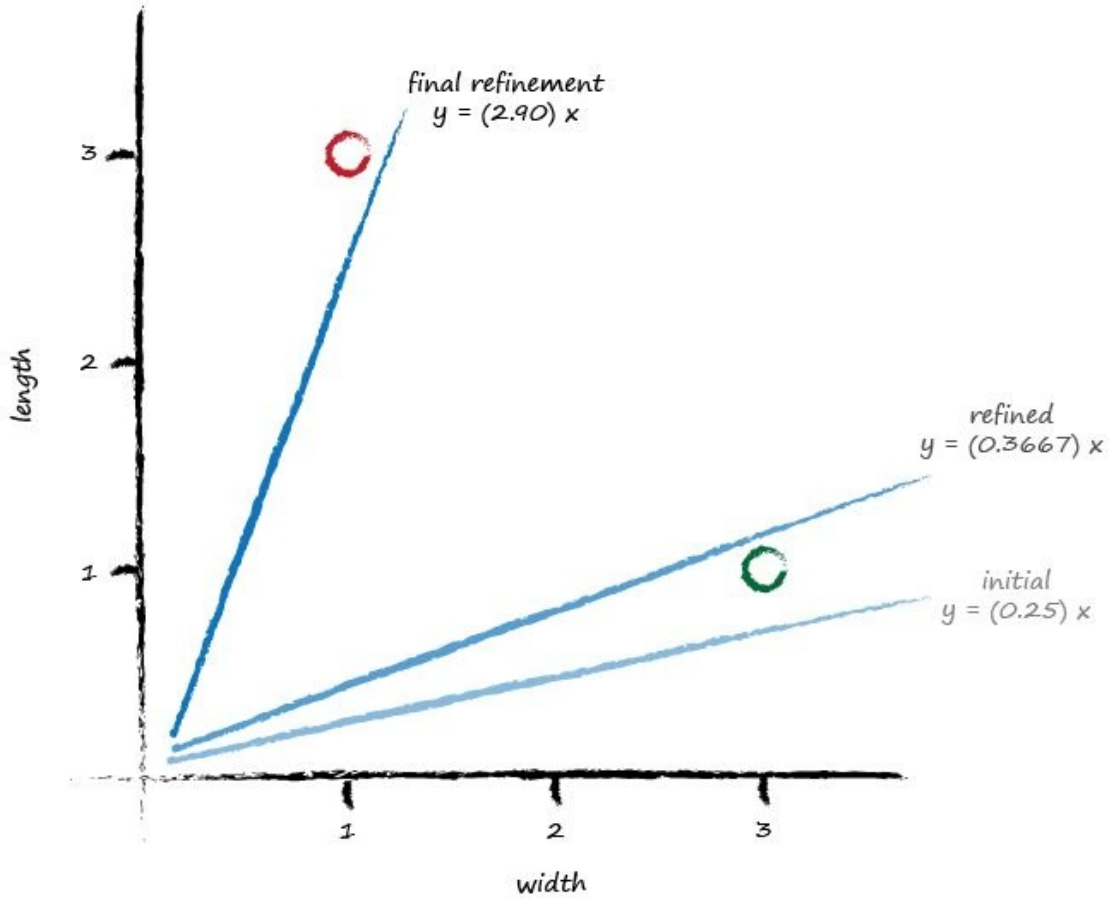
Let's see what happens when we put **x** = 1.0 into the linear function which is now using the updated **A** = 0.3667. We get **y** = 0.3667 * 1.0 = 0.3667. That's not very close to the training example with **y** = 3.0 at all.

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 Δ**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:

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

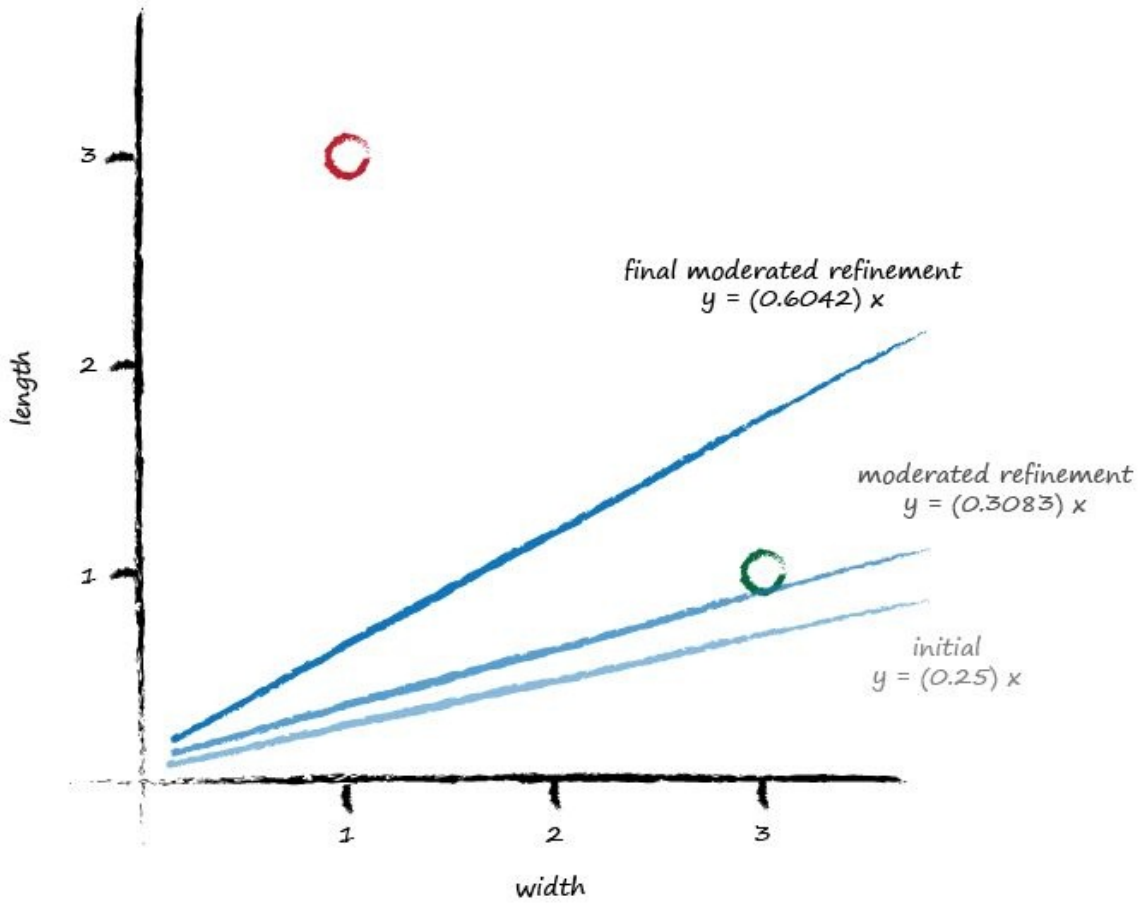
The moderating factor is often called a **learning rate**, and we've called it **L**. Let's pick **L** = 0.5 as a reasonable fraction just to get started. It simply means we only update half as much as would have done without moderation.

Running through that all again, we have an initial **A** = 0.25. The first training example gives us **y** = 0.25 * 3.0 = 0.75. A desired value of 1.1 gives us an error of 0.35. The Δ**A** = **L** (**E** / **x**) = 0.5 * 0.35 / 3.0 = 0.0583. The updated **A** is 0.25 + 0.0583 = 0.3083.

Trying out this new A on the training example at **x** = 3.0 gives **y** = 0.3083 * 3.0 = 0.9250. The line now falls on the wrong side of the training example because it is below 1.1 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 Δ**A** = **L** (**E** / **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.

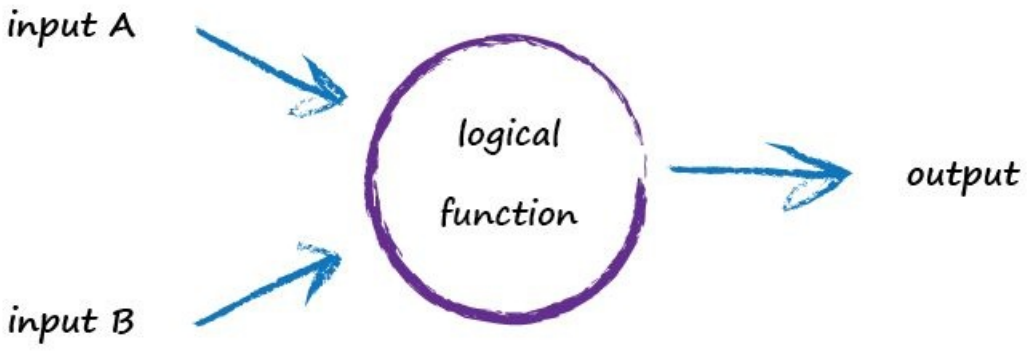
We'll be moving away from garden bugs and looking at **Boolean** logic functions. If that sounds like mumbo jumbo jargon - don't worry. George Boole was a mathematician and philosopher, and his name is associated with simple functions like AND and OR.

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

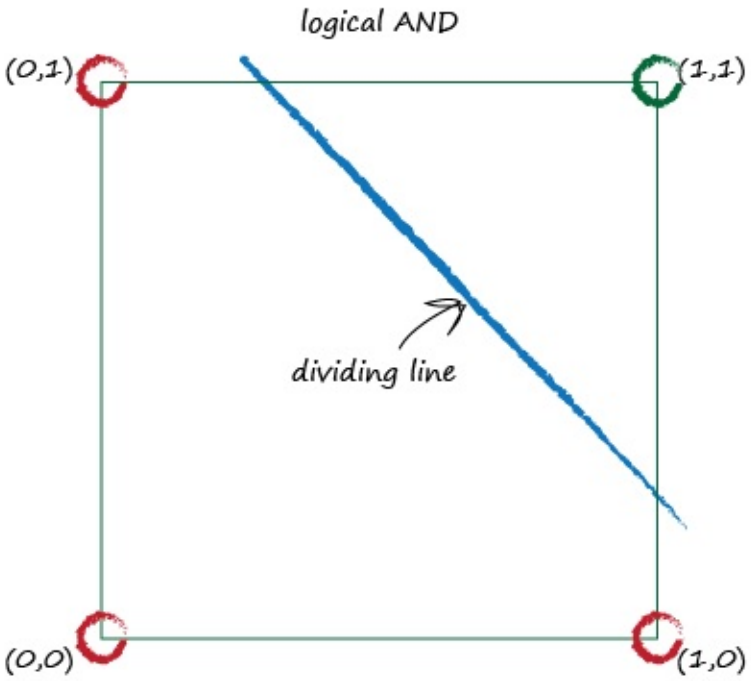
You can see quite clearly, that the AND function is only true if both A and B are true. Similarly you can see that OR is true whenever any of the inputs A or B is true.

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.

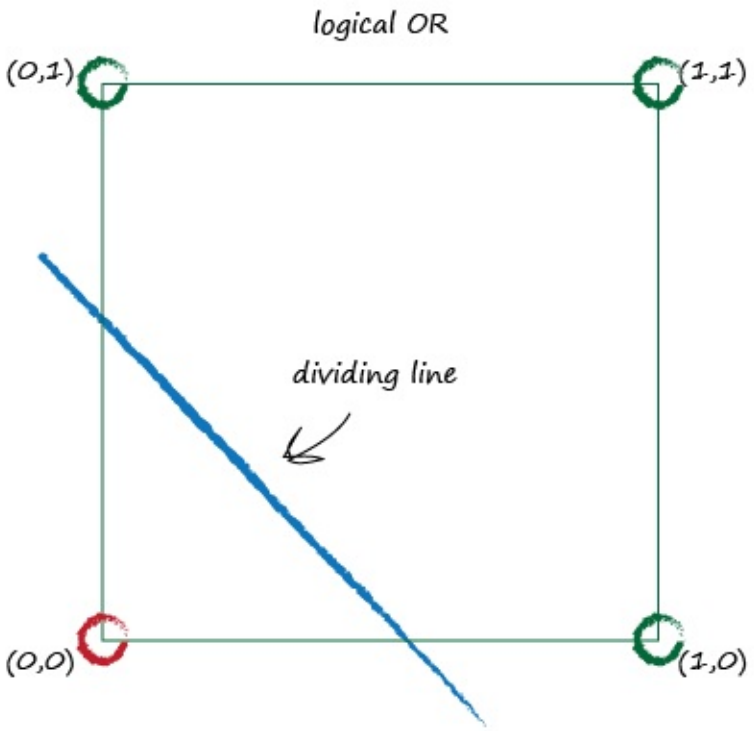


You can also see a straight line that divides the red from the green regions. That line is a linear function that a linear classifier could learn, just as we have done earlier.

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:



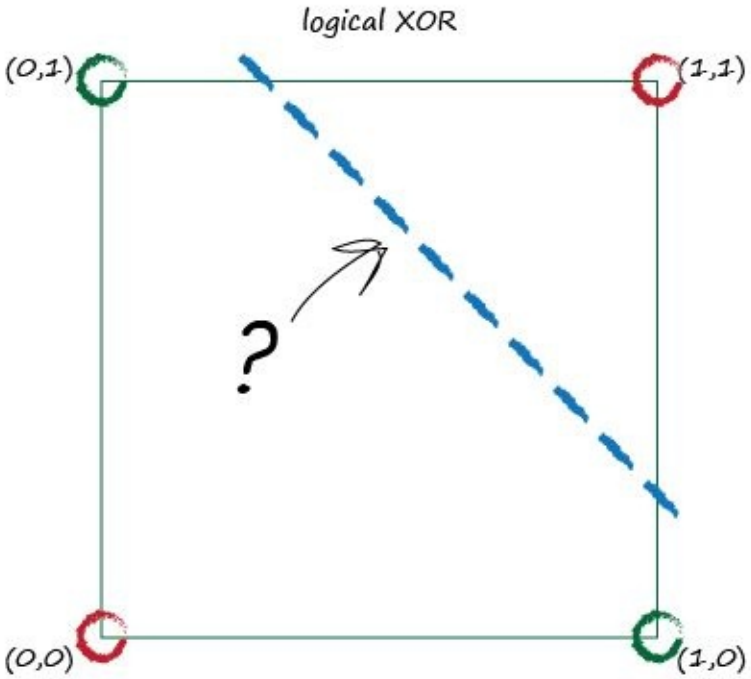
This time only the (0,0) point is red because it corresponds to both inputs A and B being false. All other combinations have at least one A or B as true, and so the output is true. The beauty of the diagram is that it makes clear that it is possible for a linear classifier to learn the Boolean OR function,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

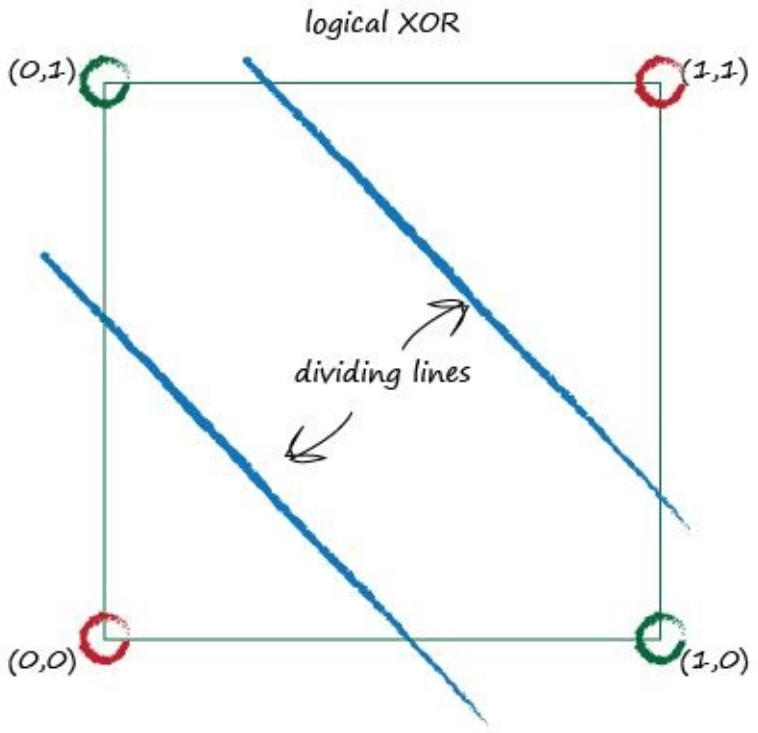
It is, in fact, impossible to have a single straight line that successfully divides the red from the green regions for the Boolean XOR. That is, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function.

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:

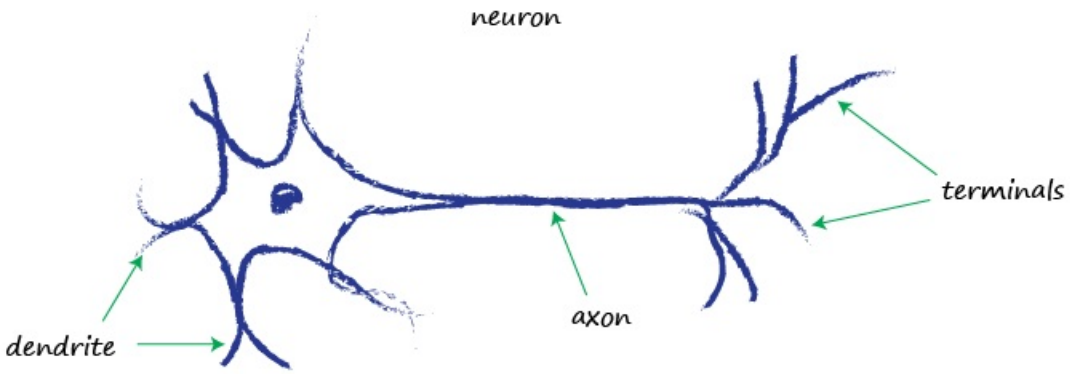
- A simple linear classifier can't separate data where that data itself isn't governed by a single linear process. For example, data governed by the logical XOR operator illustrates this.
- However the solution is easy, you just use multiple linear classifiers to divide up data that can't be separated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

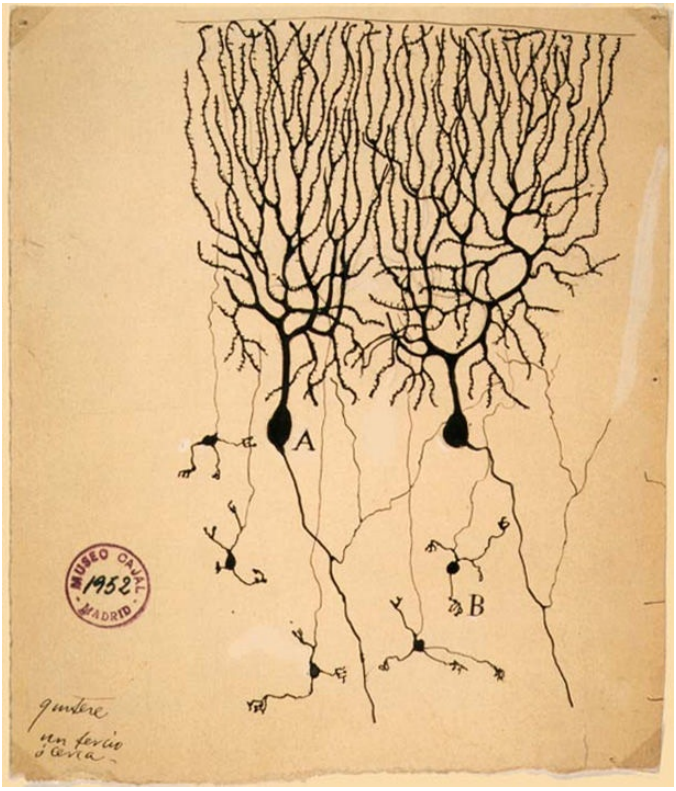
Attention turned to the architectural differences. Traditional computers processed data very much sequentially, and in pretty exact concrete terms. There is no fuzziness or ambiguity about their cold hard calculations. Animal brains, on the other hand, although apparently running at much slower rhythms,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

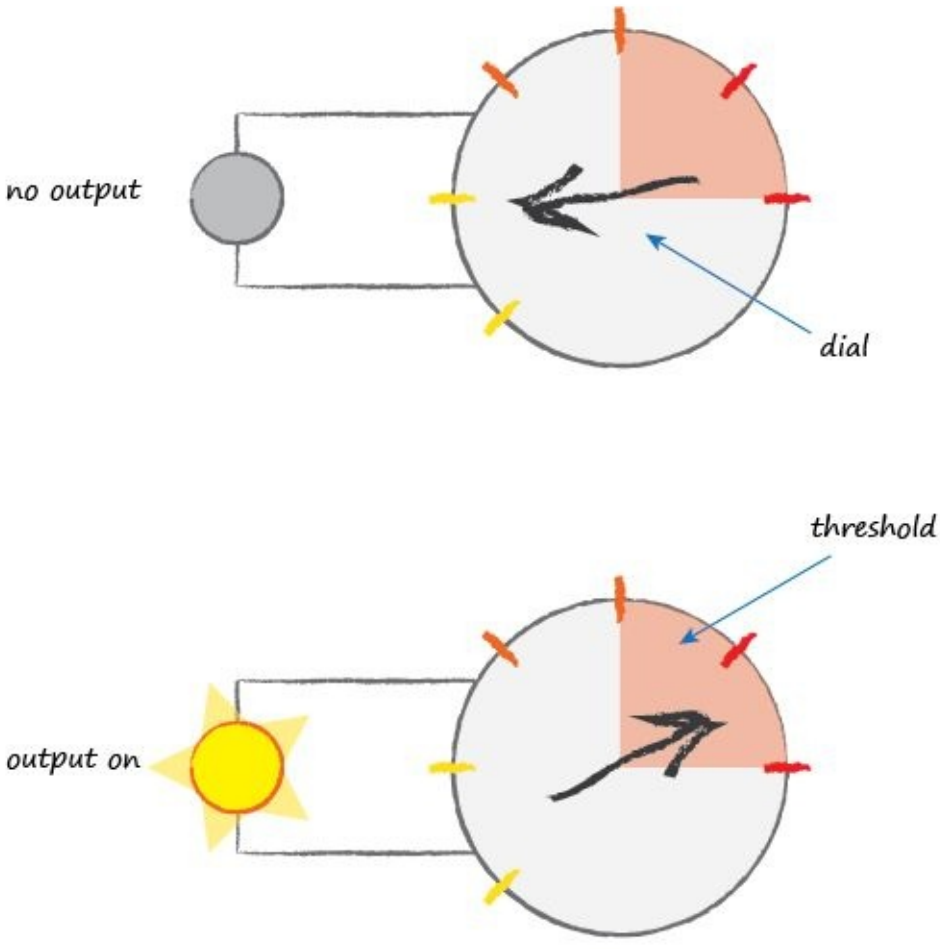
Well, the very capable human brain has about 100 billion neurons! A fruit fly has about 100,000 neurons and is capable of flying, feeding, evading danger, finding food, and many more fairly complex tasks. This number, 100,000 neurons, is well within the realm of modern computers to try to replicate. 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.

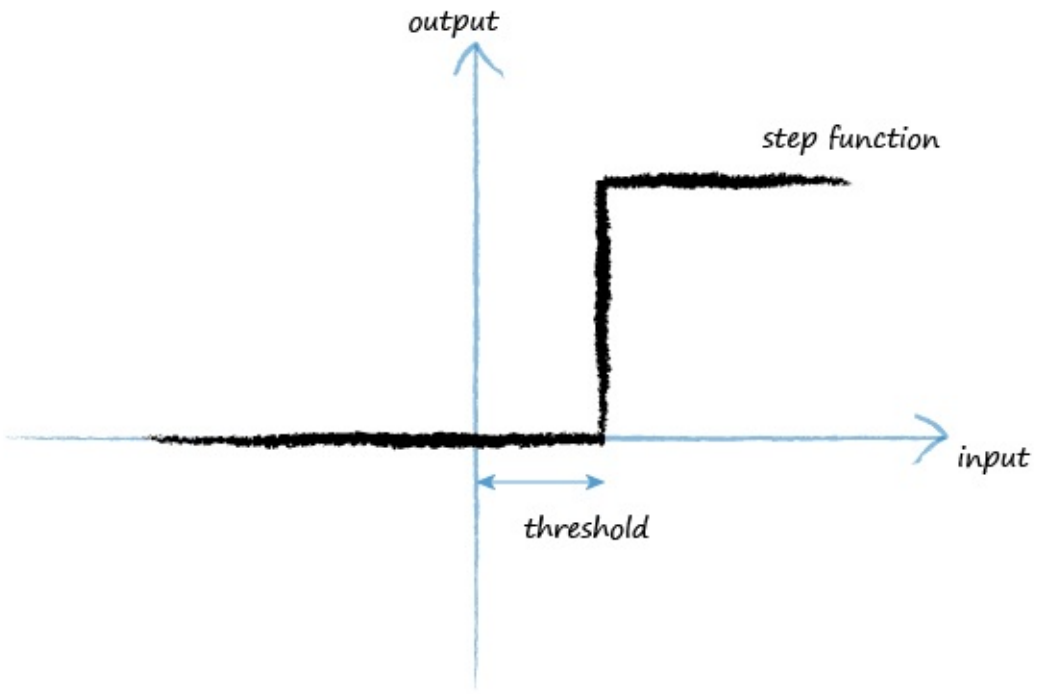
So let's look at how a neuron works. It takes an electric input, and pops out another electrical signal. This looks exactly like the classifying or predicting machines we looked at earlier, which took an input, did some processing, and popped out an output.

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**.

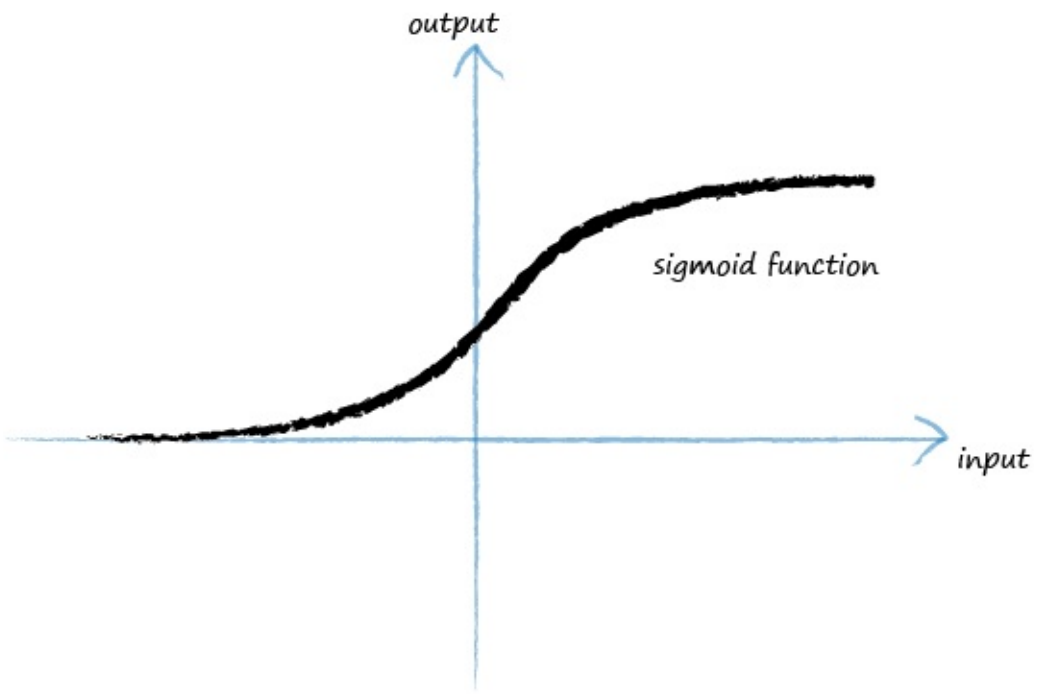


A function that takes the input signal and generates an output signal, but takes into account some kind of threshold is called an **activation function**. Mathematically, there are many such activation functions that could achieve this effect. A simple **step function** could do this:



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!

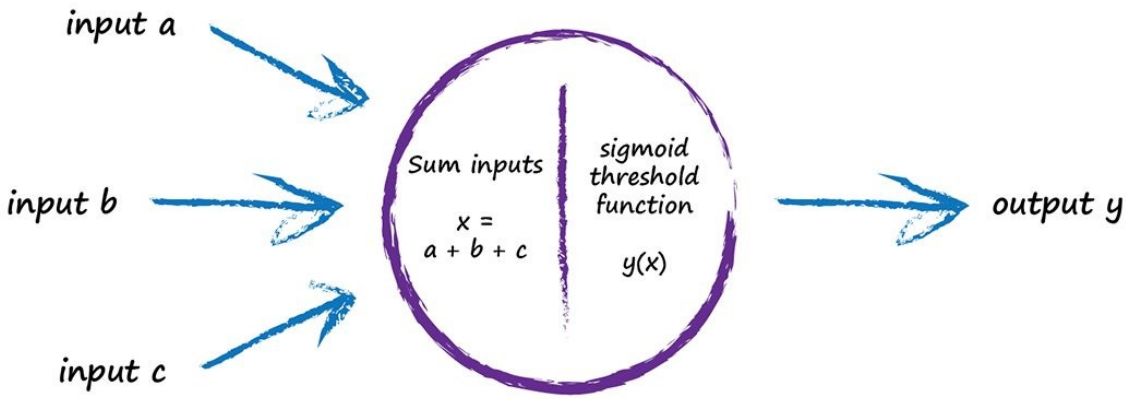
Just out of interest, when **x** is zero, $\mathbf{e^{-x}}$ is 1 because anything raised to a power of zero is 1. So **y** becomes 1 / (1 +1) or simply 1/2, a half. So the basic sigmoid cuts the y-axis at **y** = 1/2.

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.

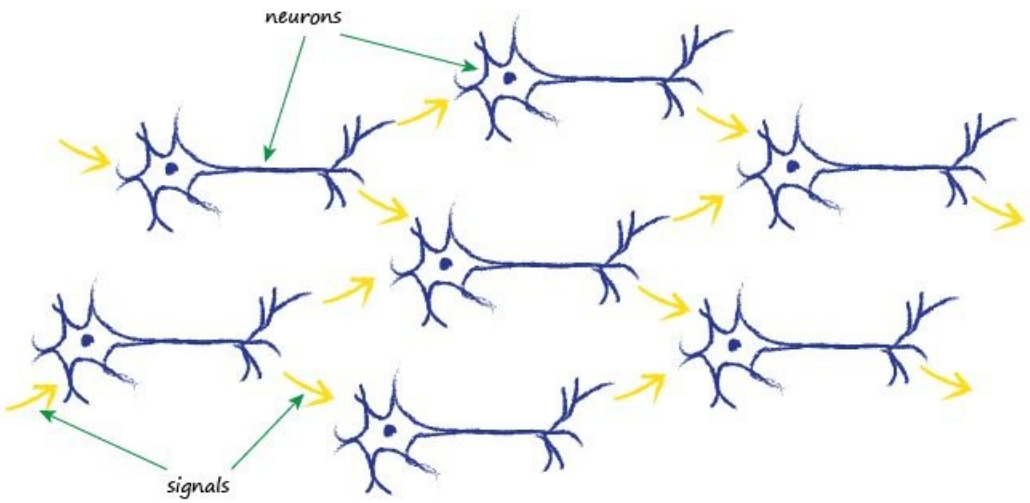
What do we do with all these inputs? We simply combine them by adding them up, and the resultant sum is the input to the sigmoid function which controls the output. This reflects how real neurons work. The following diagram illustrates this idea of combining inputs and then applying the threshold to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

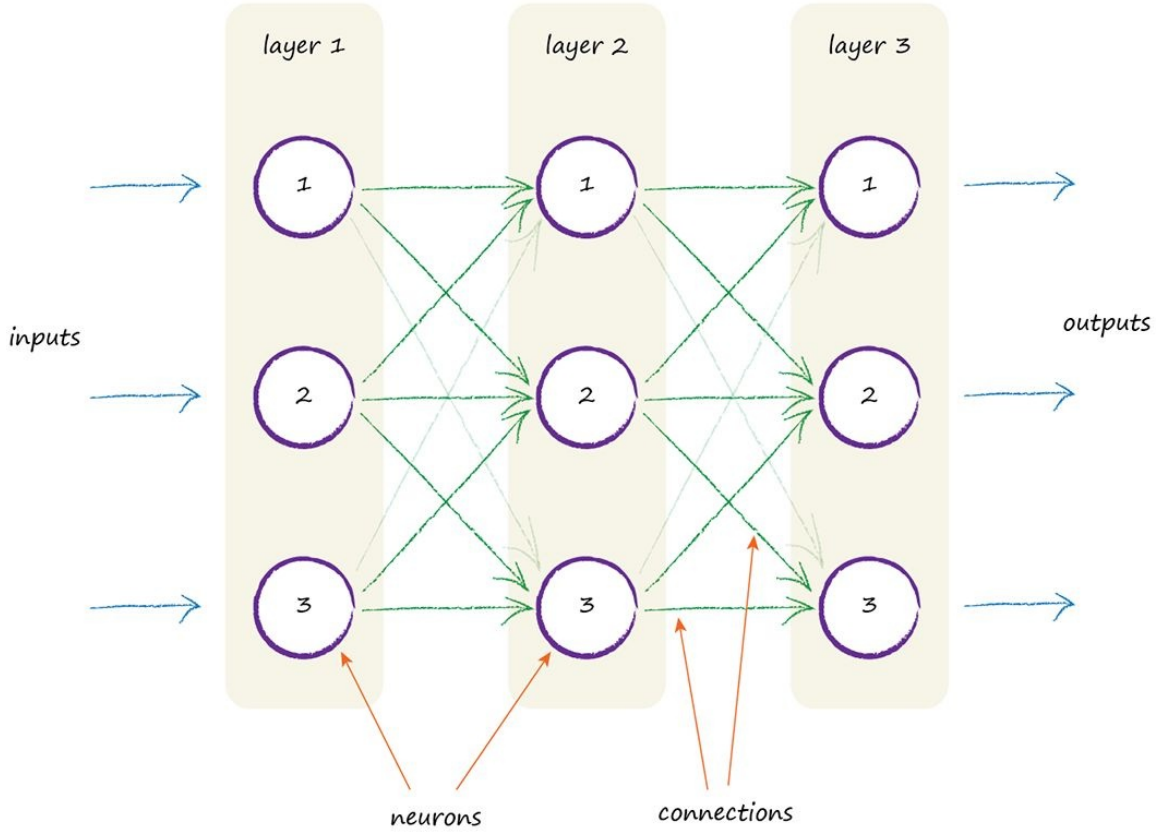
but not quite, large enough because when combined the signal is large enough to overcome the threshold. In an intuitive way, this gives you a sense of the more sophisticated, and in a sense fuzzy, calculations that such neurons can do.

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:

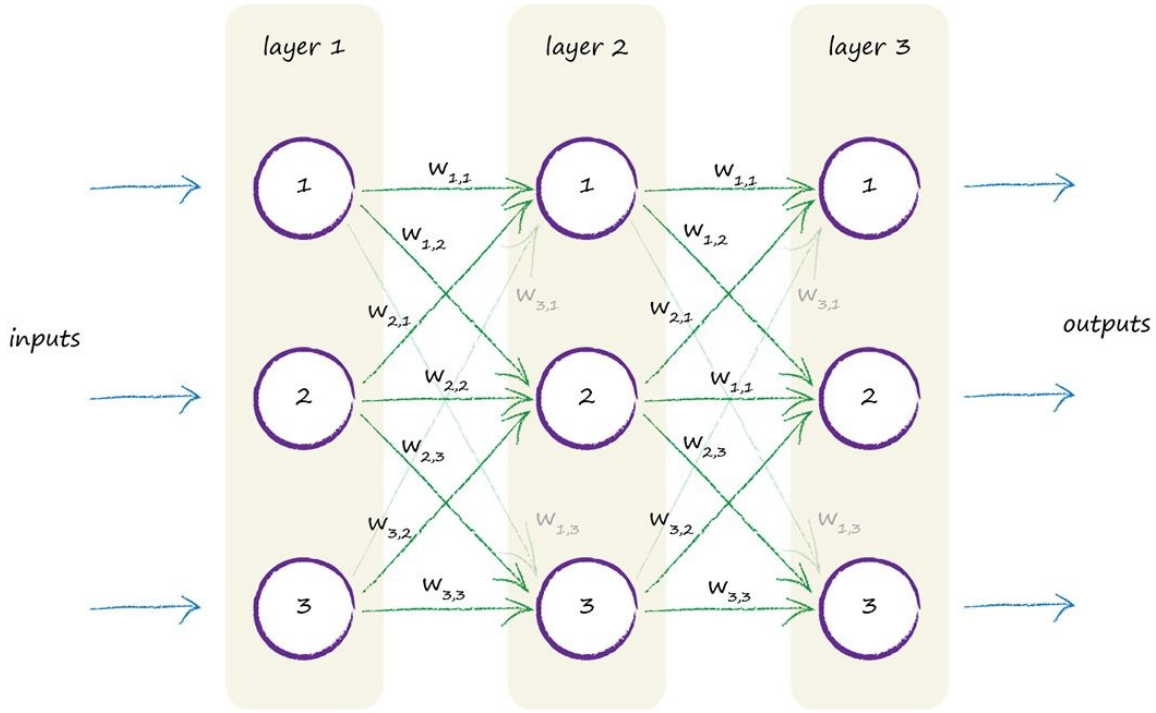


You can see the three layers, each with three artificial neurons, or **nodes**. You can also see each node connected to every other node in the preceding and next layers.

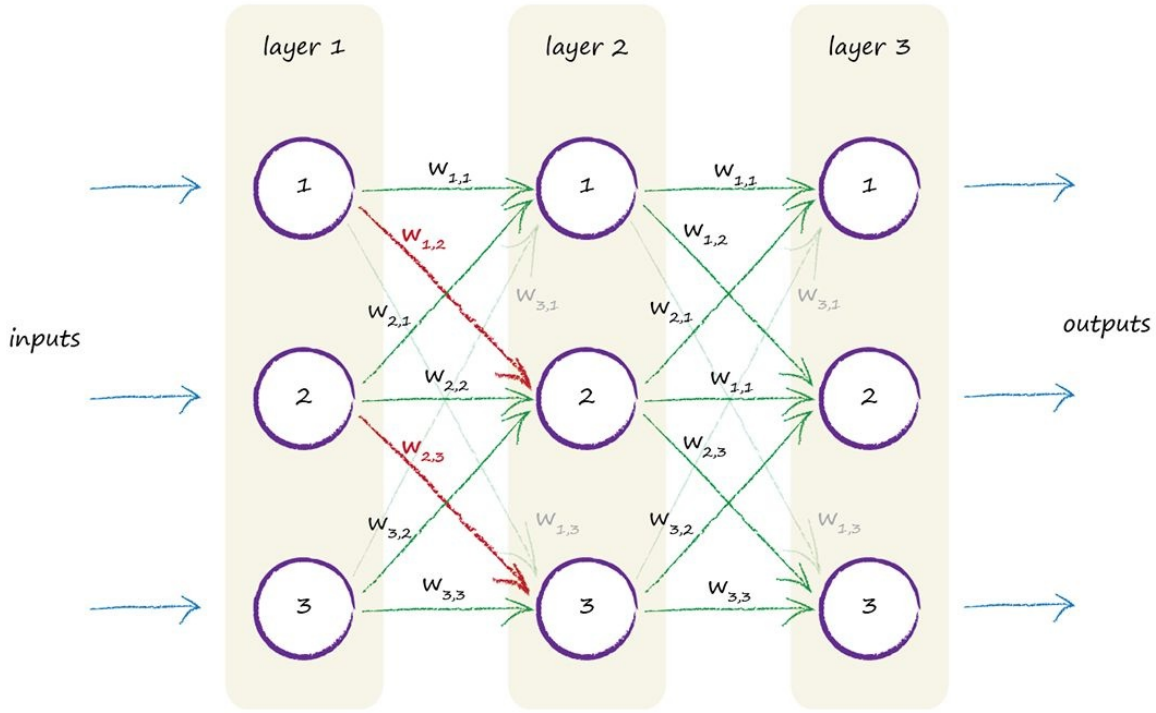
That's great! But what part of this cool looking architecture does the learning? What do we adjust in response to training examples? Is there a parameter that we can refine like the slope of the linear classifiers we looked at earlier?

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.

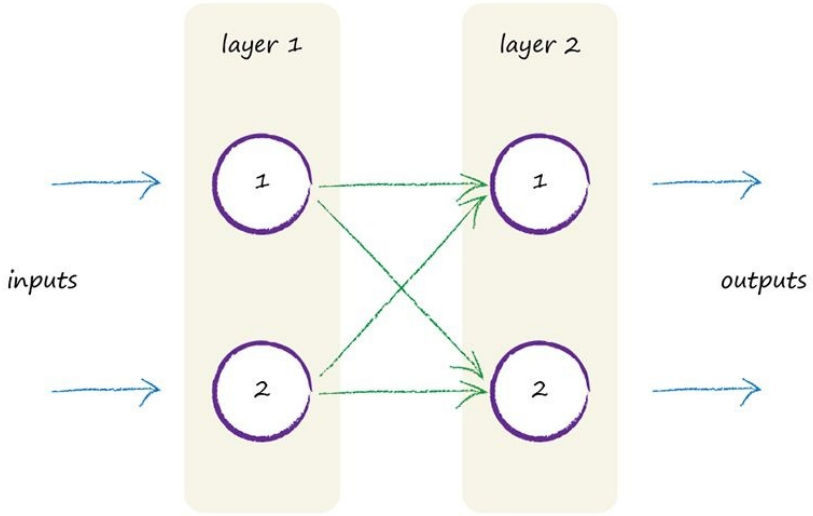
● Biological brains, made of connected neurons, are the inspiration for artificial neural networks.

Following Signals Through A Neural Network

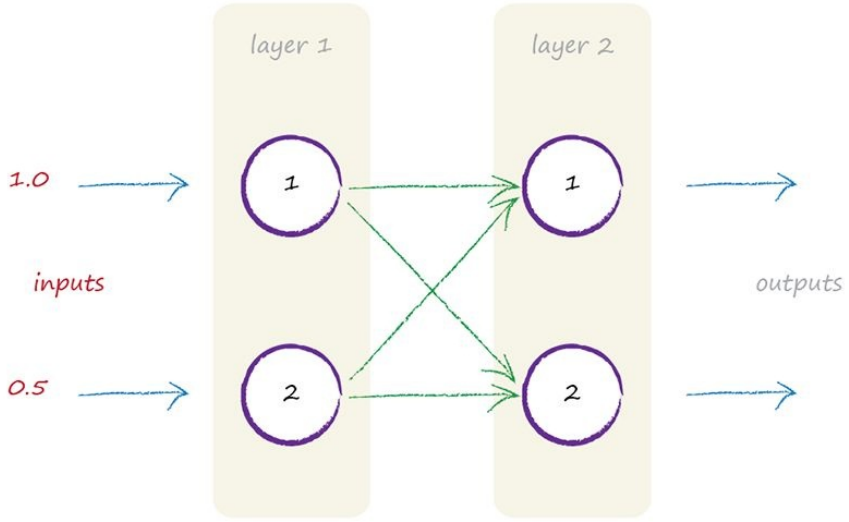
That picture of 3 layers of neurons, with each neuron connect to every other in the next and previous layers, looks pretty amazing.

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.



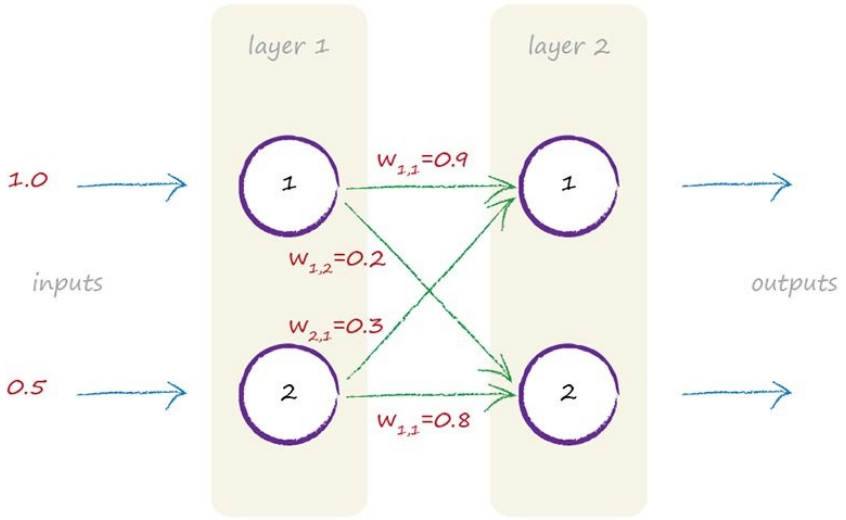
Just as before, each node turns the sum of the inputs into an output using an activation function. We’ll also use the sigmoid function $y = \frac{1}{(1+e^{-x})}$ that we saw before, where x is the sum of incoming signals to a neuron, and **y** is the output of that neuron.

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.

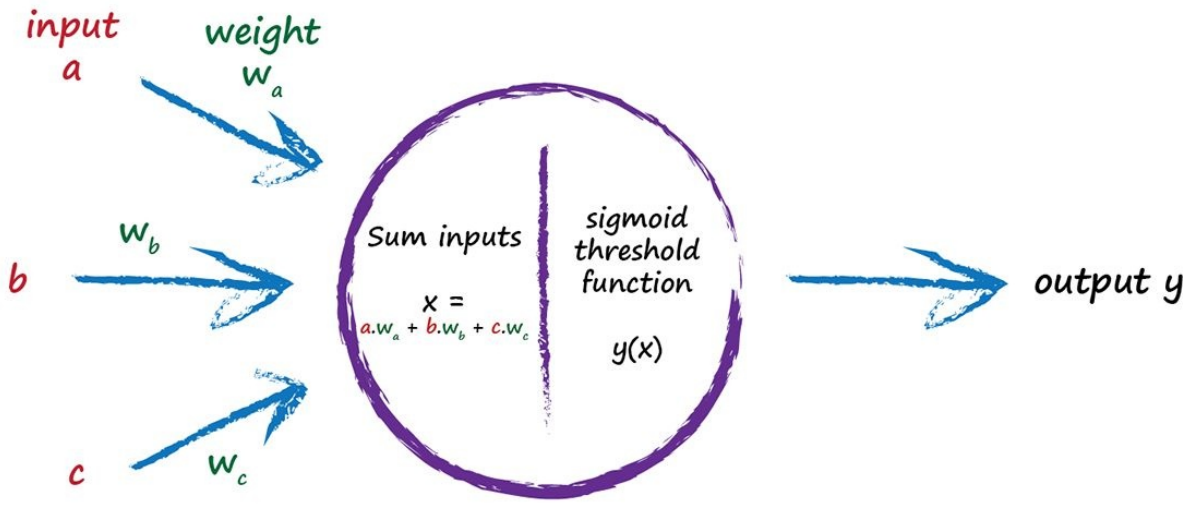


Let’s start calculating!

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)

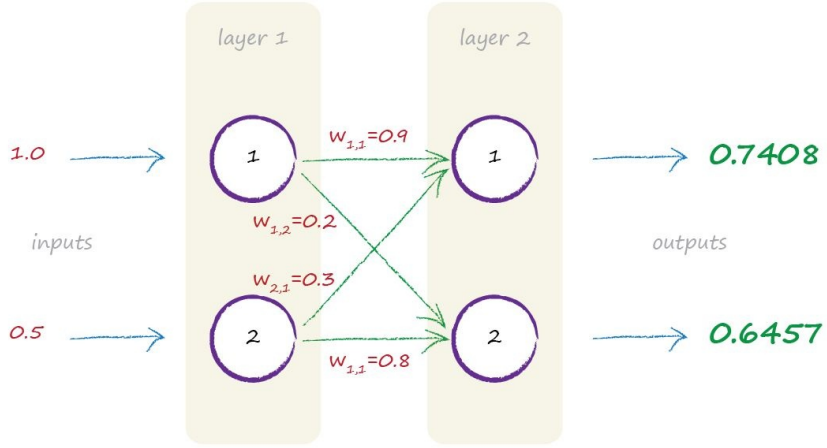
x = (1.0 * 0.2) + (0.5 * 0.8)

x = 0.2 + 0.4

x = 0.6

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

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



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!

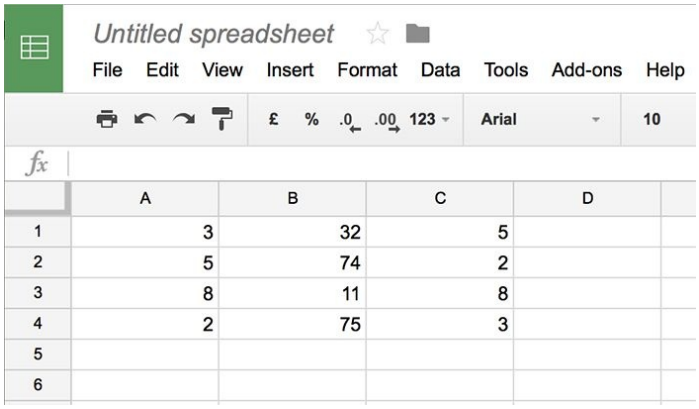
So how can matrices help? Well, they help us in two ways. First, they allow us to compress writing all those calculations into a very simple short form. That's great for us humans, because we don't like doing a lot of work because it's boring, and we're prone to errors anyway. The second benefit is that 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.



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.

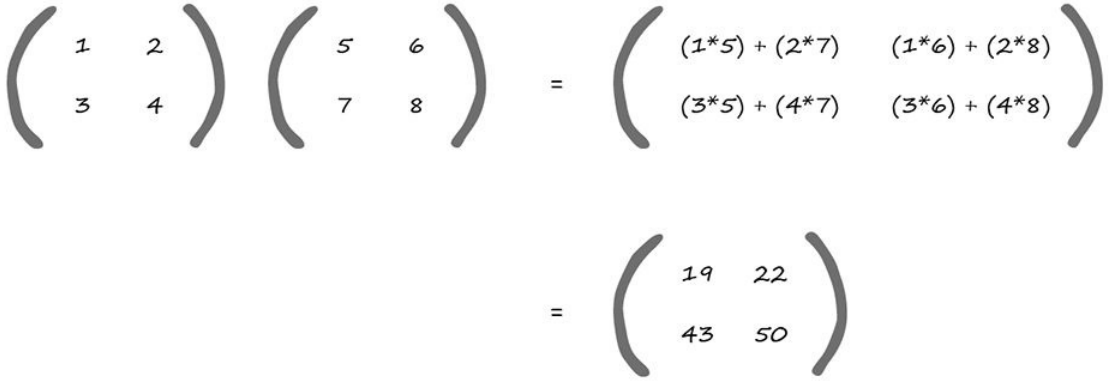
Also, some people use square brackets around matrices, and others use round brackets like we have.

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.

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

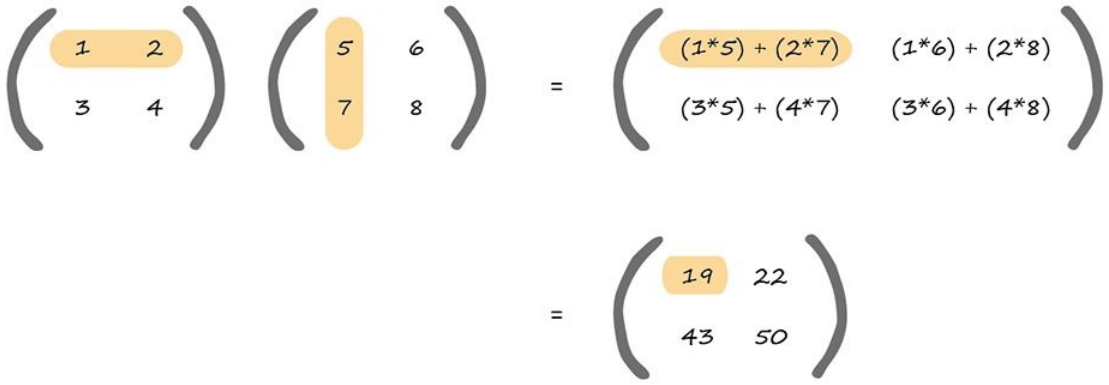
Now, matrices become useful to us when we look at how they are multiplied. You may remember how to do this from school, but if not we'll look at it again.

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



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.

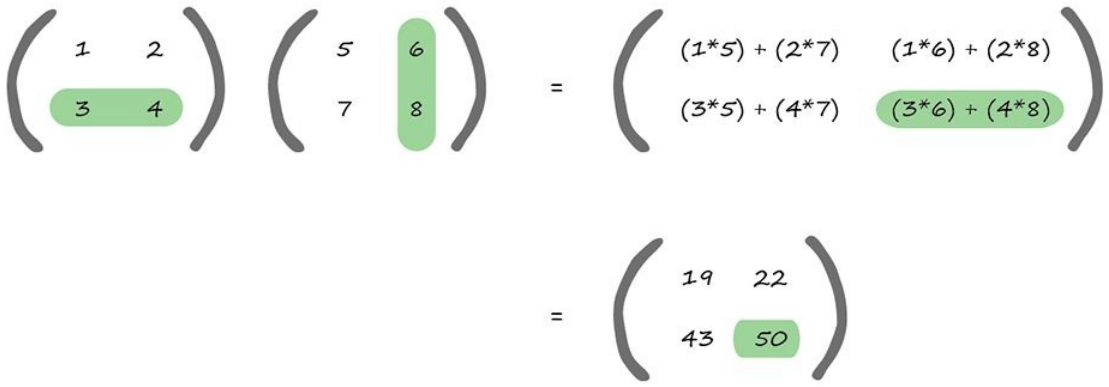
Instead, matrices are multiplied using different rules. You may be able to work them out by looking at the above example. If not, have a look at the following which highlights how the top left element of the answer is worked out.



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.

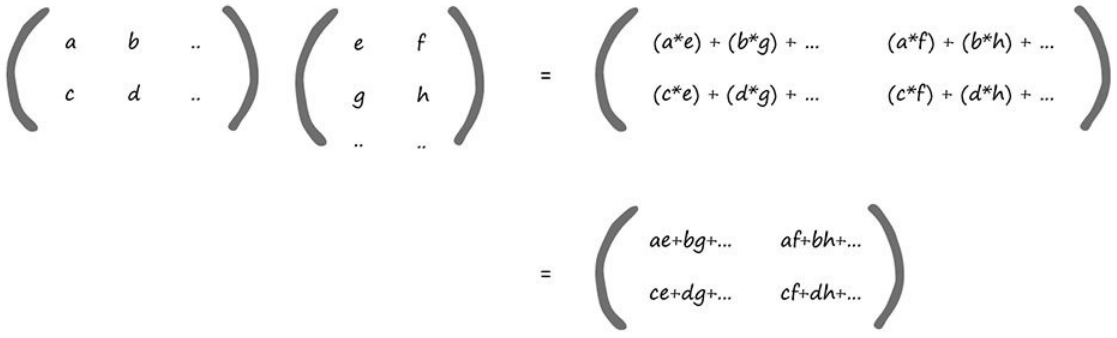
That's a lot of words, but it is easier to see it in action. Have a go yourself. The following explains how the bottom right element is worked out.



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!

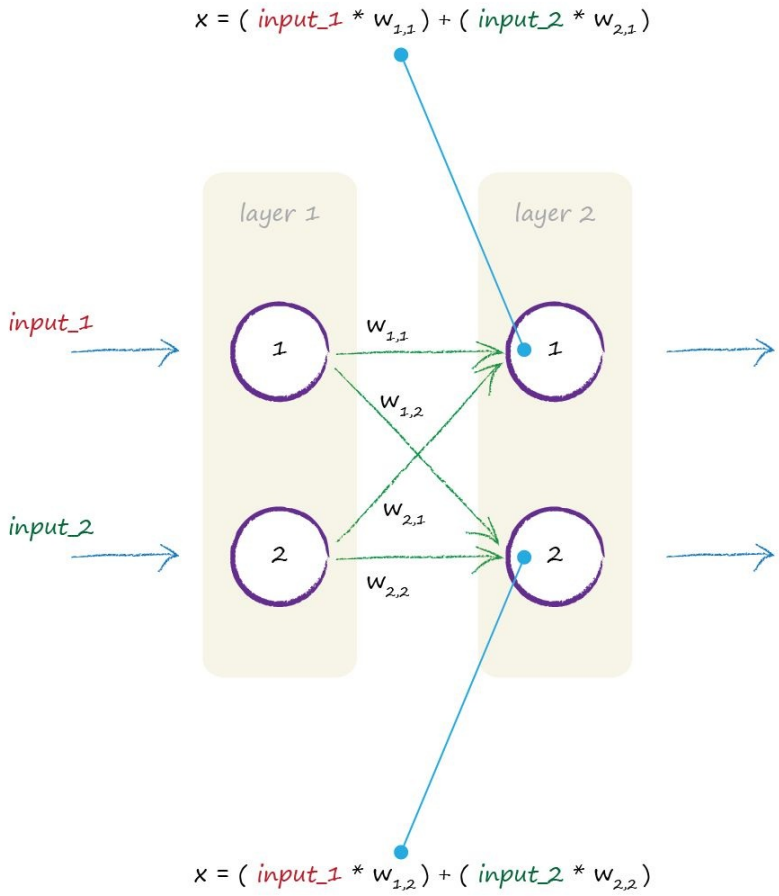
Look what happens if we replace the letters with words that are more meaningful to our neural networks. The second matrix is a two by one matrix, but the multiplication approach is the same.

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

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 **X** = **W**·**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

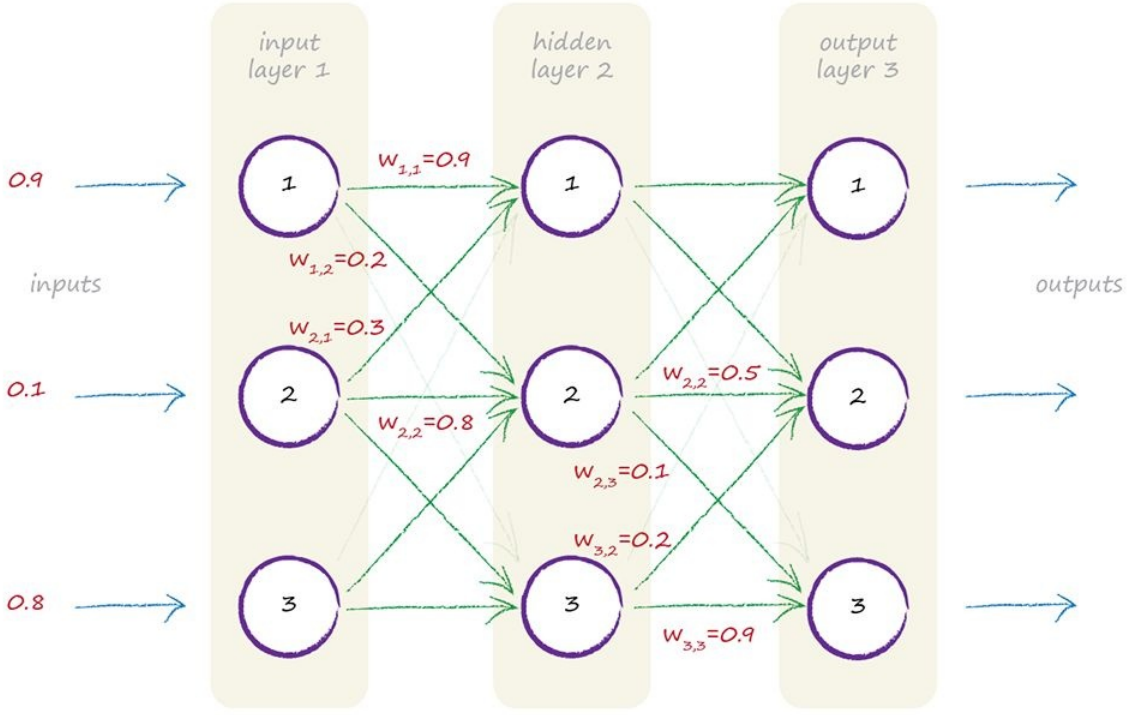
us to write, no matter the size of neural network.

● 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 **X** = **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.

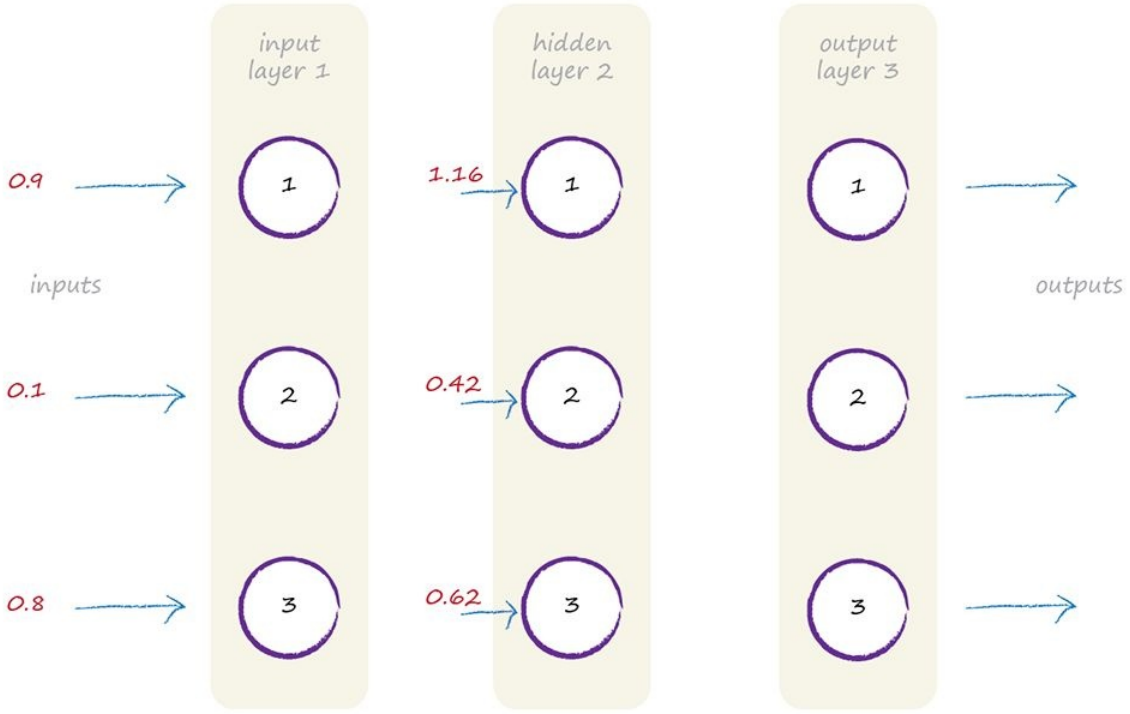
$$X_{hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

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

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.

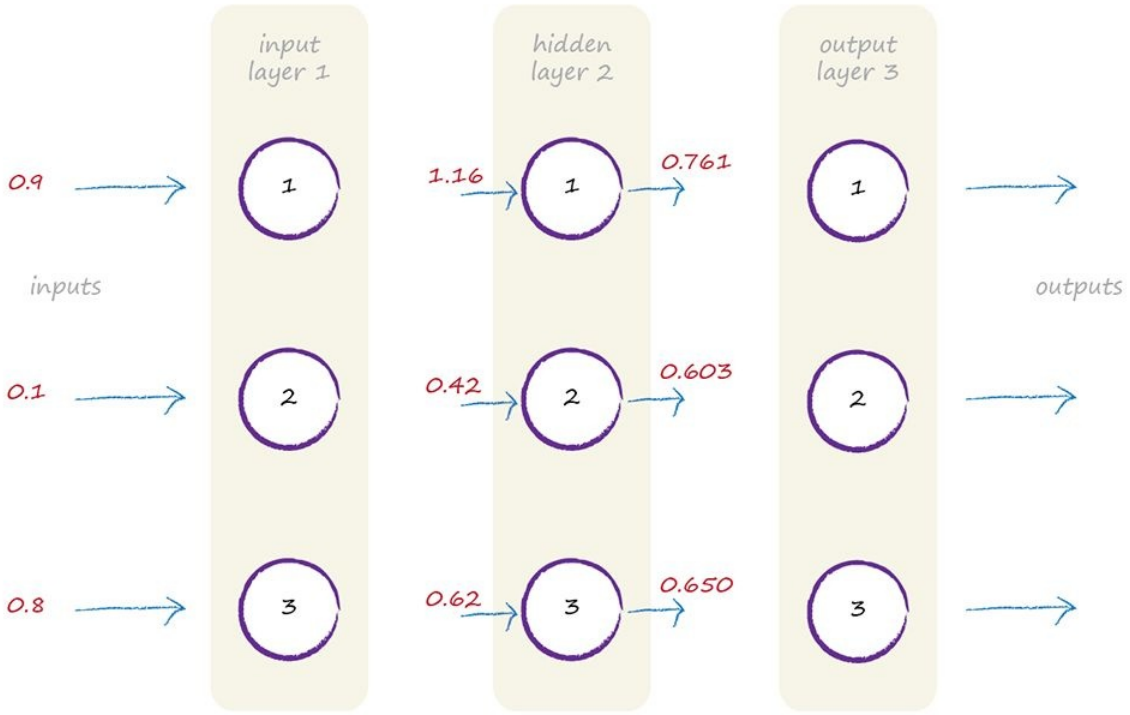
$$O_{hidden} = \text{signmoid} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

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

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:

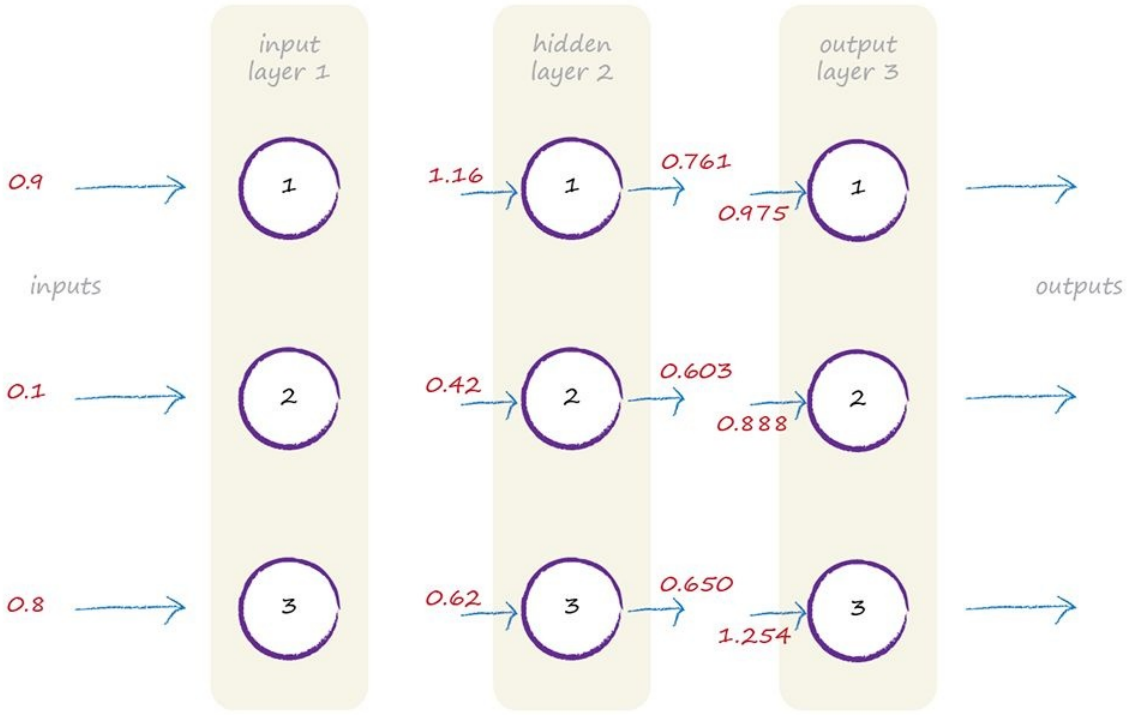
$$X_{output} = W_{hidden\_output} \cdot O_{hidden}$$

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.

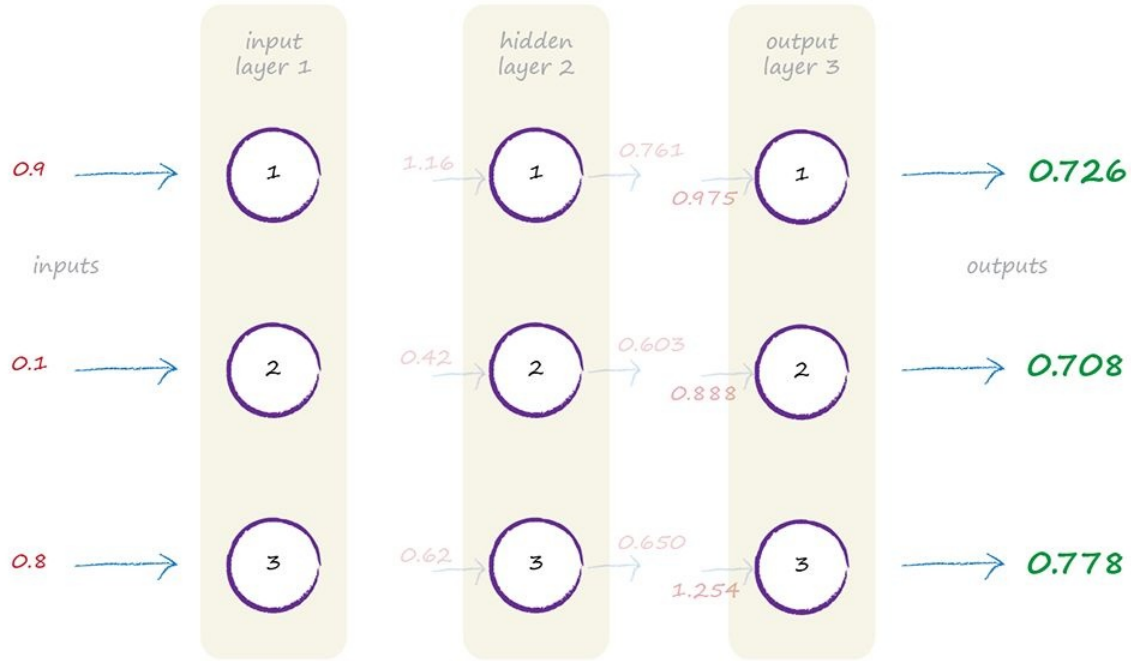


All that remains is to apply the sigmoid activation function, which is easy.

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

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

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

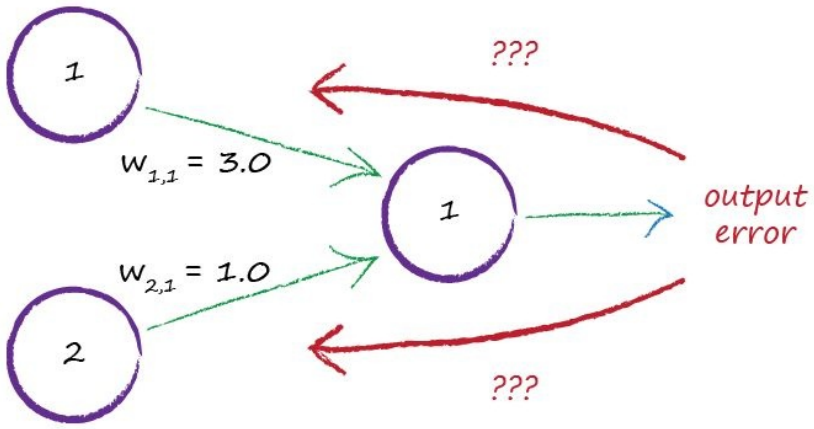
neural network itself so that it improves its outputs.

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.

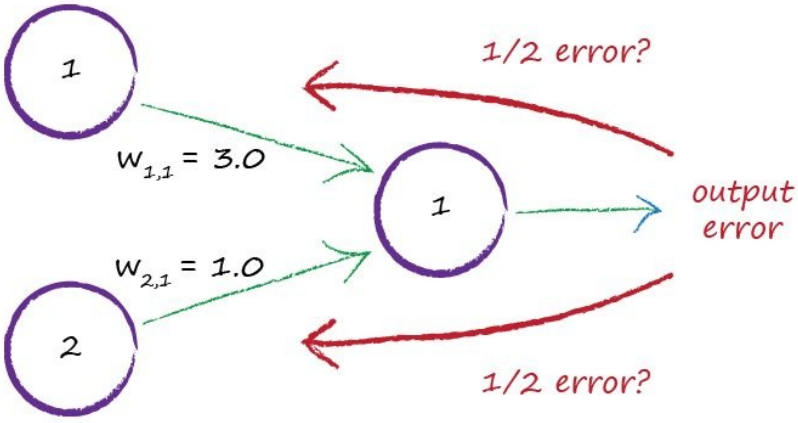


Things were much simpler when we just had one node feeding into an output node. If we have two nodes, how do we use that output error?

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.

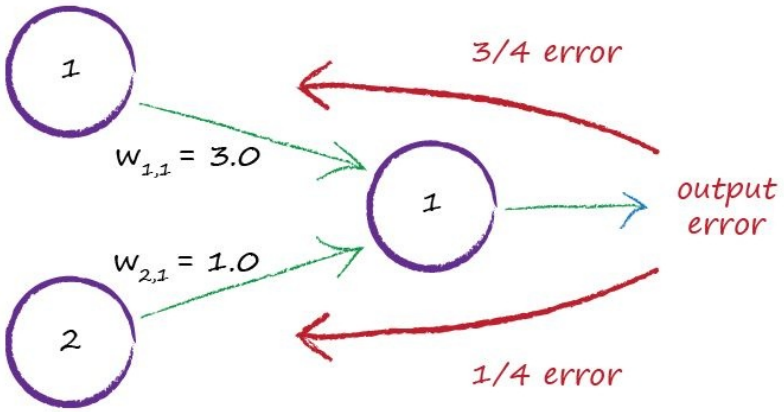
There is a tiny chance that only one link of many was responsible for the error, but that chance is extremely miniscule. If we did change a weight that was already "correct", making it worse, it would get improved during the next iterations so all is not lost.

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

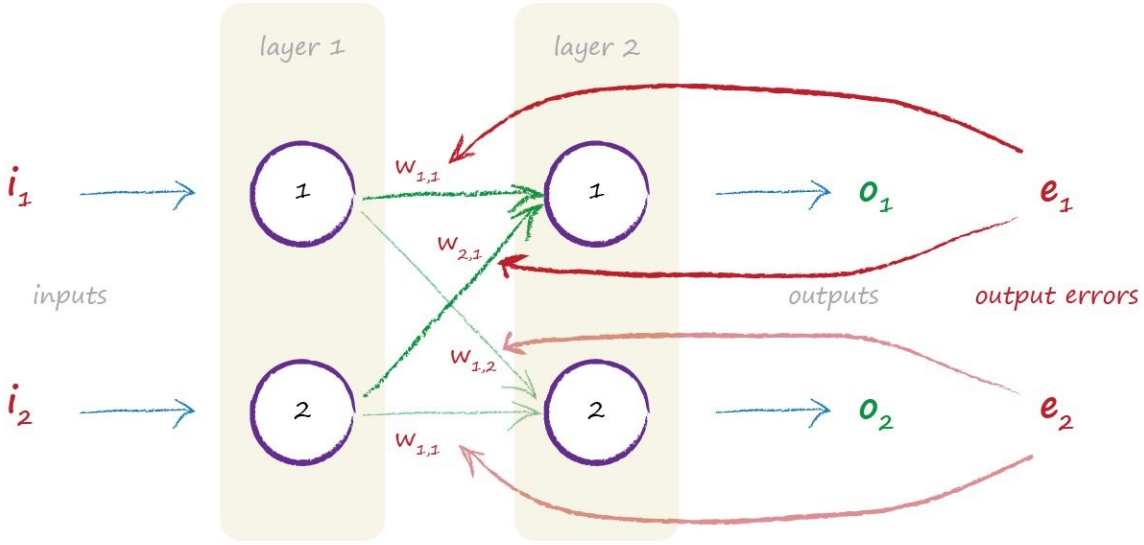
that output node in **proportion** to each link's contribution to the error, indicated by the size of the link's weight.

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.

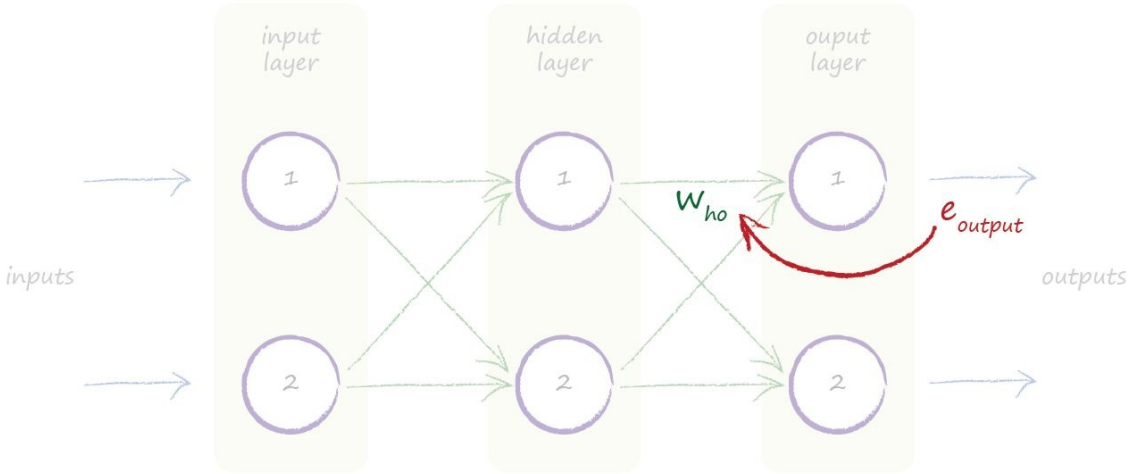
If the weights were equal, the fractions will both be half, as you'd expect. Let's see this just to be sure. Let's say $\mathbf{w_{11}}$=4 and $\mathbf{w_{21}}$=4, then the fraction is 4/(4+4) = 4/8 = 1/2 for both cases.

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?

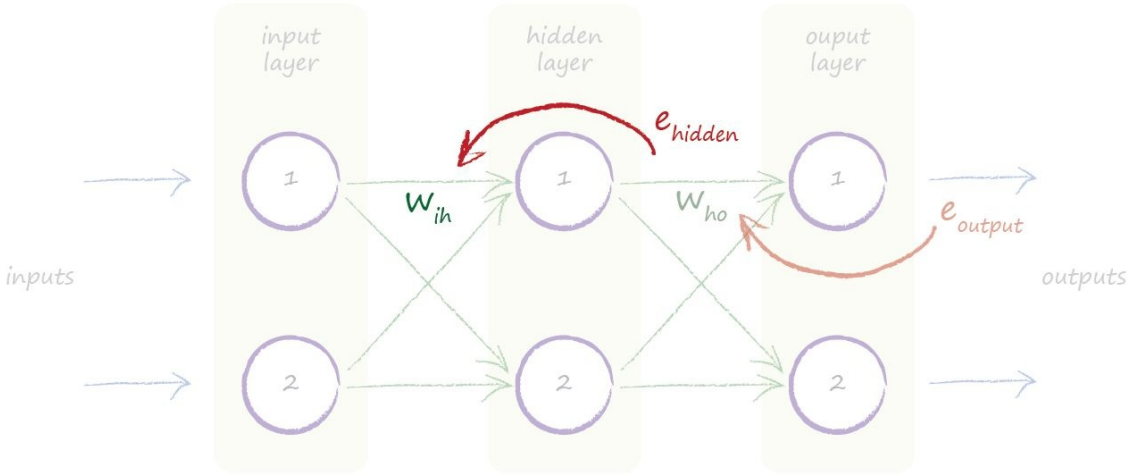
Backpropagating Errors To More Layers

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.

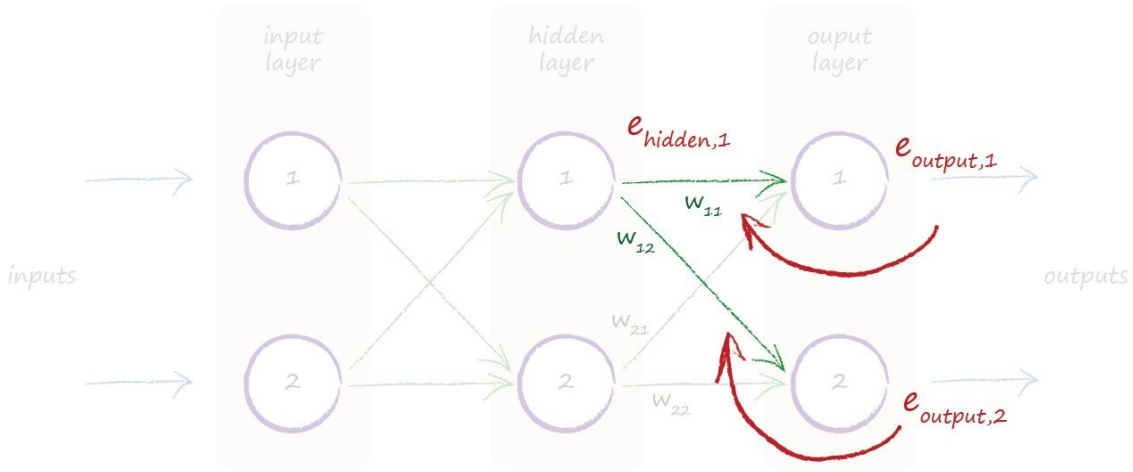
By showing this visually, we can see what we need to do for the new additional layer. We simply take those errors associated with the output of the hidden layer nodes $\mathbf{e_{hidden}}$, and split those again proportionately across the preceding links between the input and hidden layers $\mathbf{w_{ih}}$.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

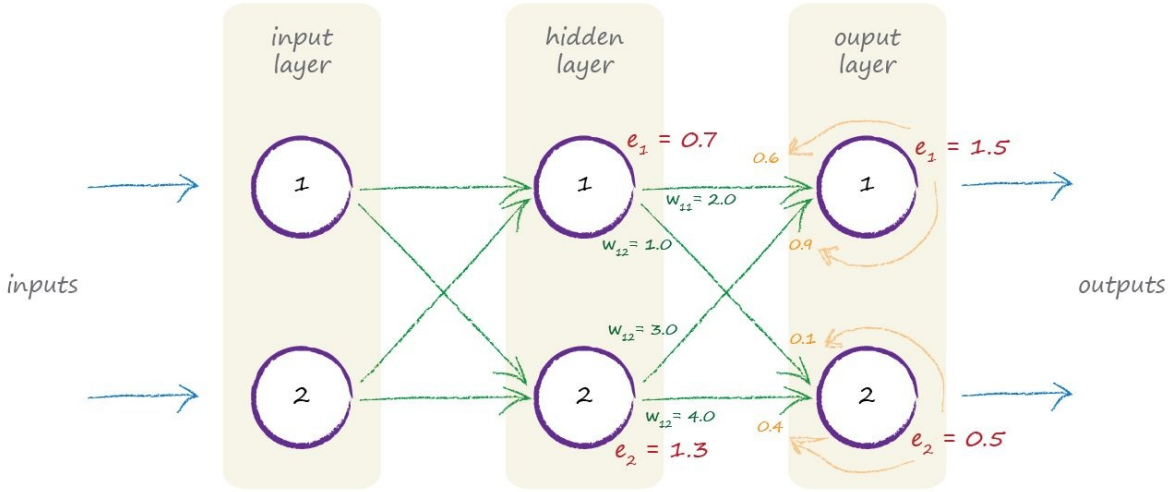
we have a fraction of the output error $\mathbf{e_{output,1}}$ on the link with weight $\mathbf{w_{11}}$ and also a fraction of the output error $\mathbf{e_{output,2}}$ from the second output node on the link with weight $\mathbf{w_{12}}$.

So let's write this down.

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

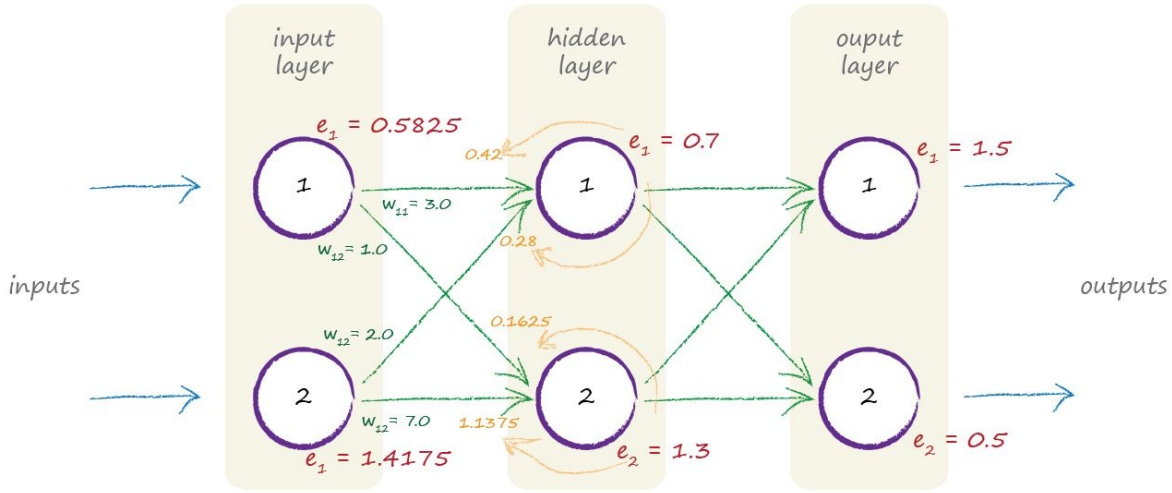
$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

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.

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

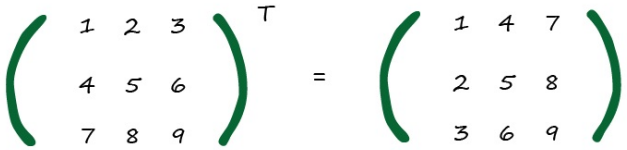
Some can spot the matrix multiplication easily, others will take a little longer staring at this expression! Here it is:

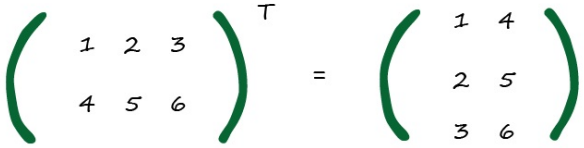
$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

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**.

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

Yikes! Let's not untangle that.

Instead of trying to be too clever, we could just simply try random combinations of weights until we find a good one?

That's not always such a crazy idea when we're stuck with a hard problem. The approach is called a **brute force** method. Some people use brute force methods to try to crack passwords, and it can work if your password is an English word and not too long, because there aren't too many for a fast home computer to work through. Now, imagine that each weight could have 1000 possibilities between -1 and +1, like 0.501, -0.203 and 0.999 for example. Then for a 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.

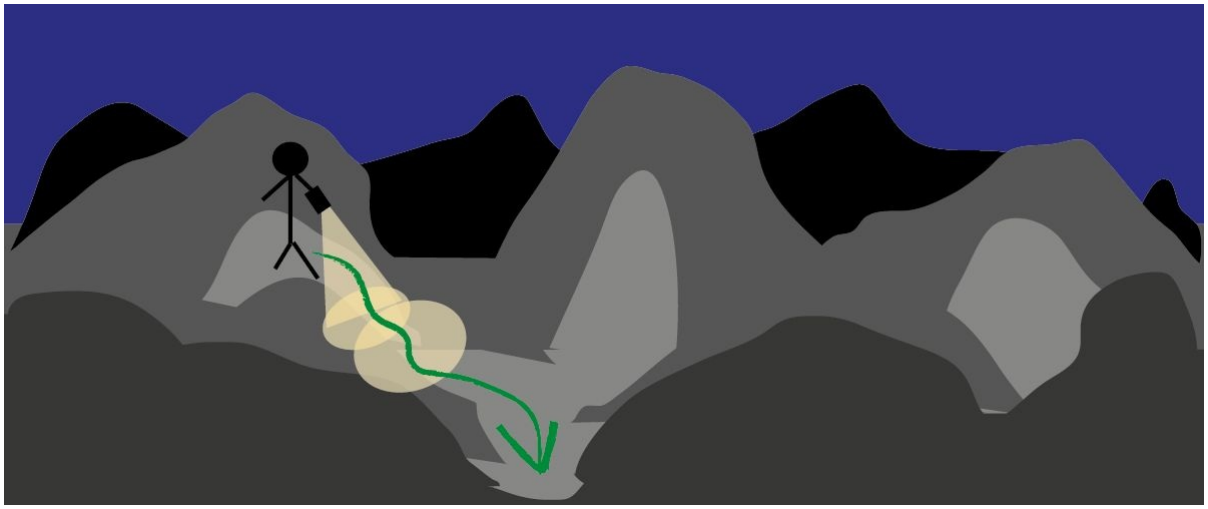
The first thing we must do is embrace **pessimism**.

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

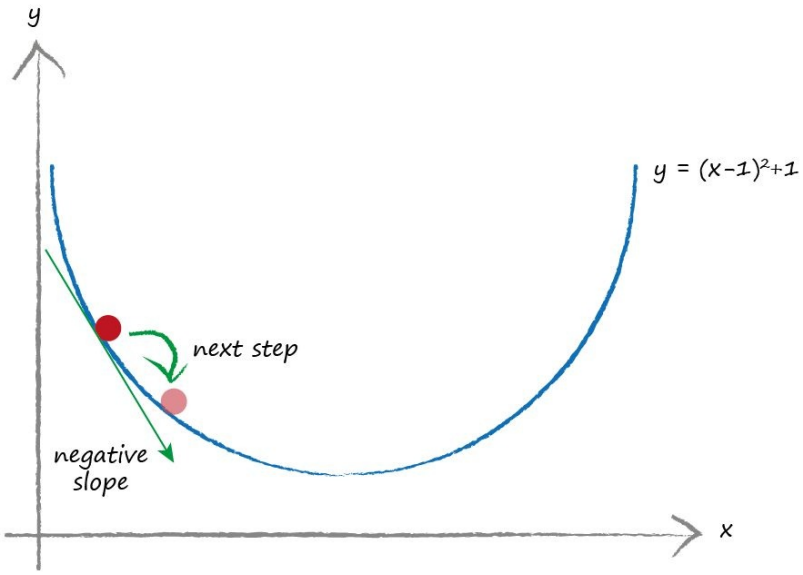
Now imagine that complex landscape is a mathematical function. What this gradient descent method gives us is an ability to find the minimum without actually having to understand that complex function enough to work it out mathematically. If a function is so difficult that we can't easily find the minimum 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.

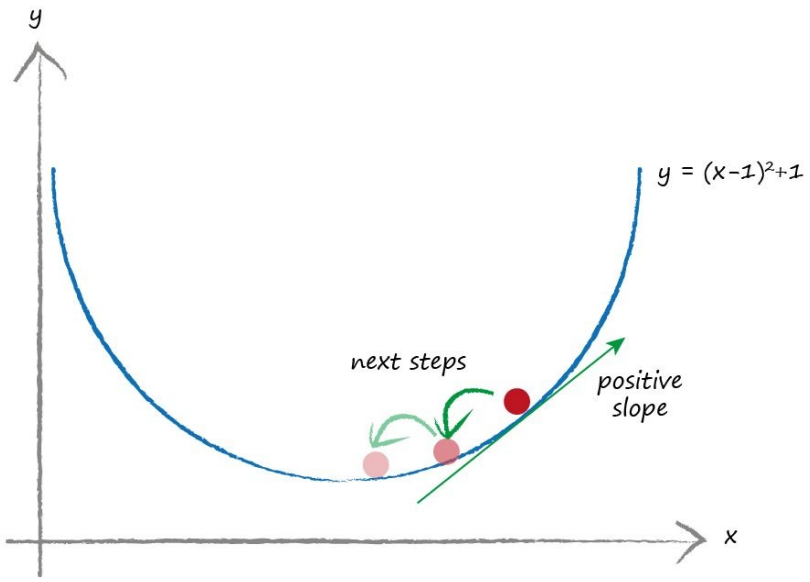
The following graph shows a simple function $\mathbf{y} = (\mathbf{x}\text{-}1)^2 + 1$. If this was a function where y was the error, we would want to find the **x** which minimises it.

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.

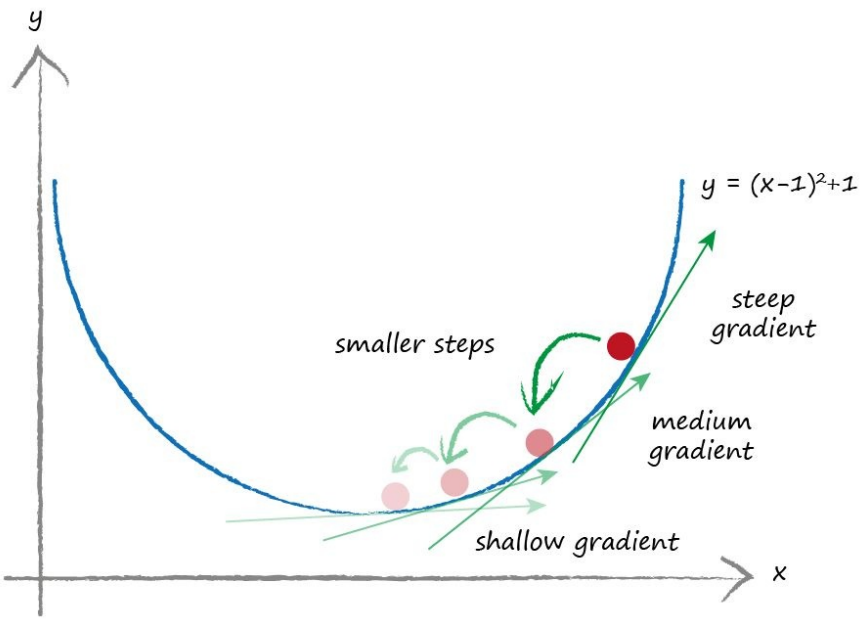
Let's imagine we started somewhere else, as shown in the next graph.



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.

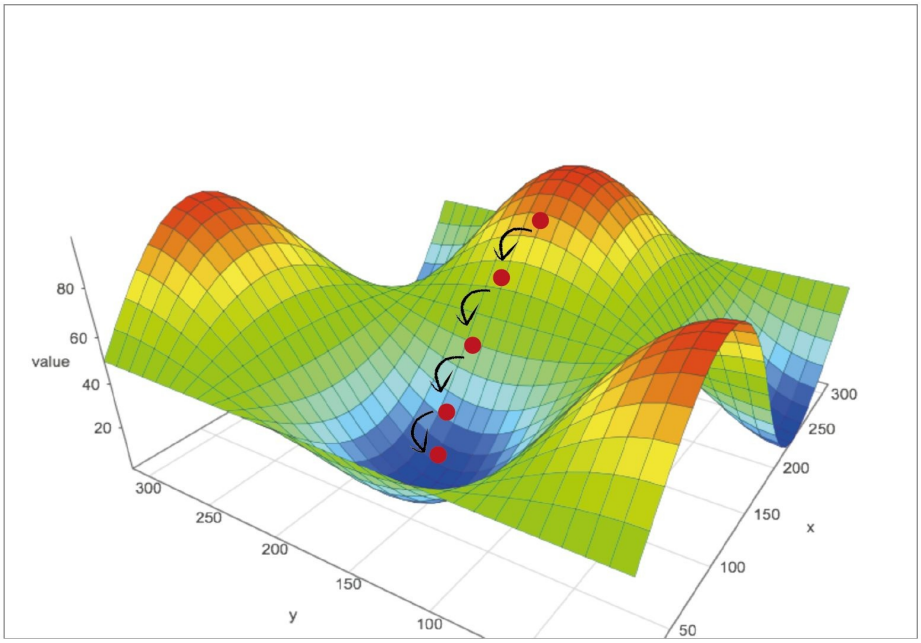


By the way, did you notice that we increase **x** in the opposite direction to the gradient? A positive gradient means we reduce **x**. A negative gradient means we increase **x**. The graphs make this clear, but it is easy to forget and get it the wrong way around.

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!

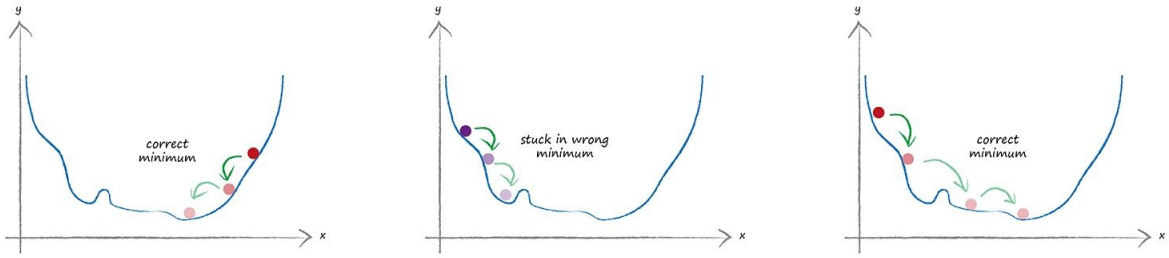
The following again illustrates gradient descent but with a slightly more complex function that depends on 2 parameters. This can be represented in 3 dimensions with the height representing the value of the function.



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:

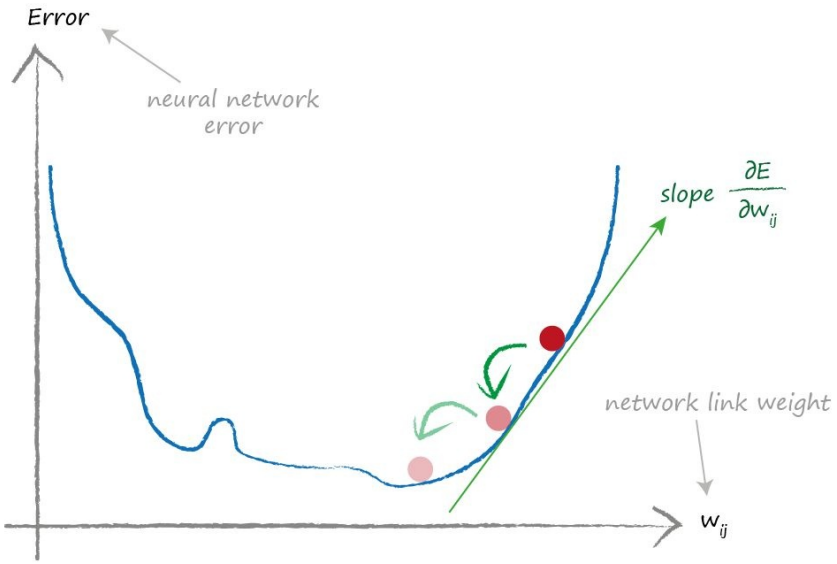
- The algebra needed to work out the slope for gradient descent is easy enough with this squared error.
- The error function is smooth and continuous making gradient descent work well - there are no gaps or abrupt jumps.
- The gradient gets smaller nearer the minimum, meaning the risk of overshooting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

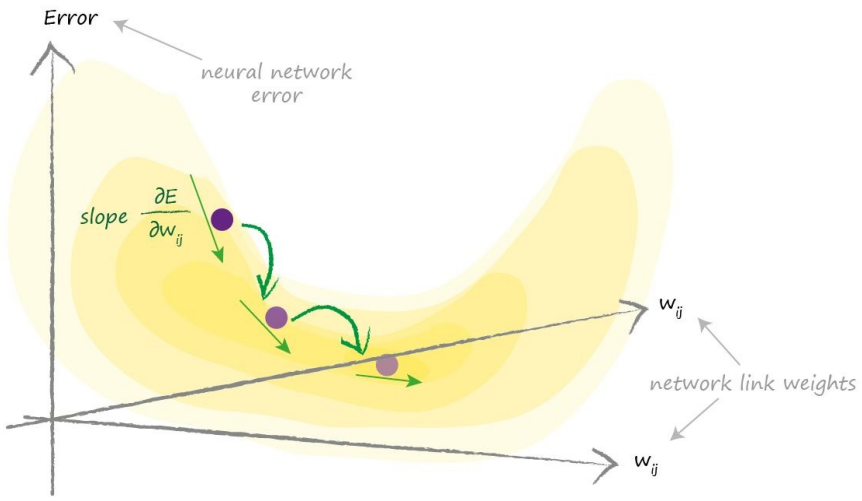
To do gradient descent, we now need to work out the slope of the error function with respect to the weights. This requires **calculus**. You may already be familiar with calculus, but if you're not, or just need a reminder, the **Appendix** contains a gentle introduction. Calculus is simply a mathematically 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.



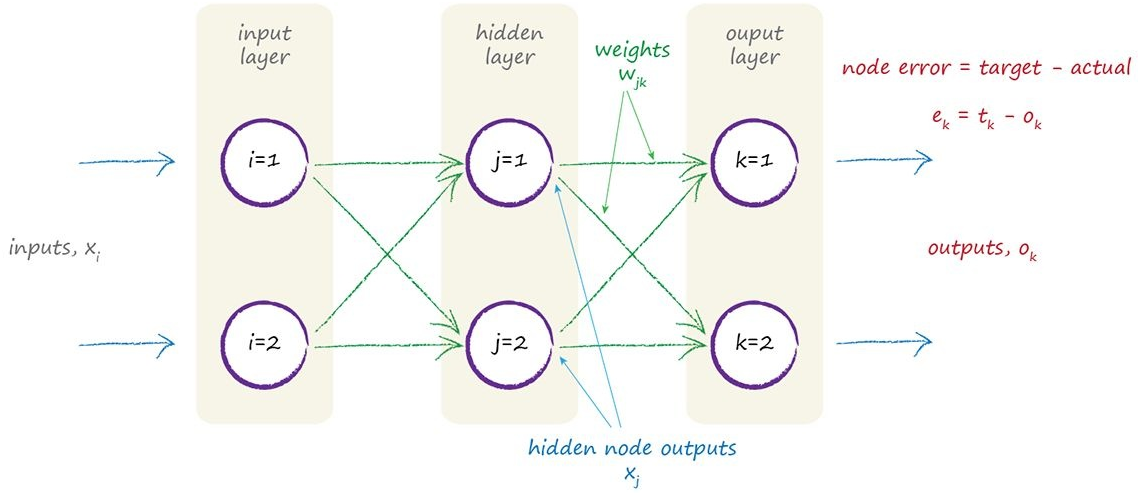
It's harder to visualise that error surface as a function of many more parameters, but the idea to use gradient descent to find the minimum is still the same.

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.

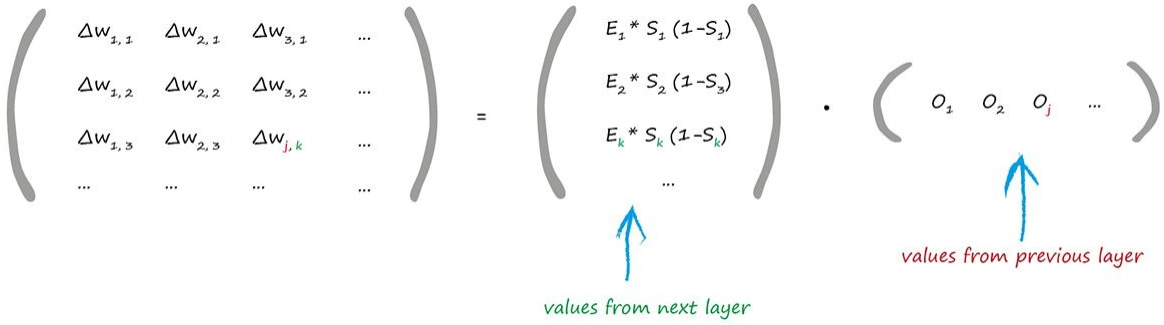
$$new\ w_{jk} = old\ w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

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 **α**, 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.



I've left out the learning rate **α** 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.

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

That's it! Job done.

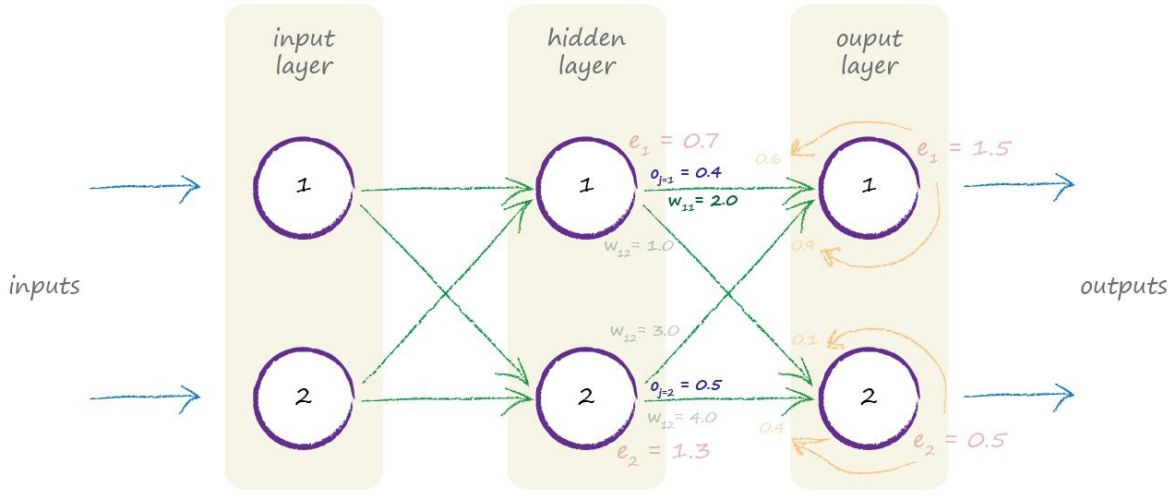
Key Points:

- 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.

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

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.

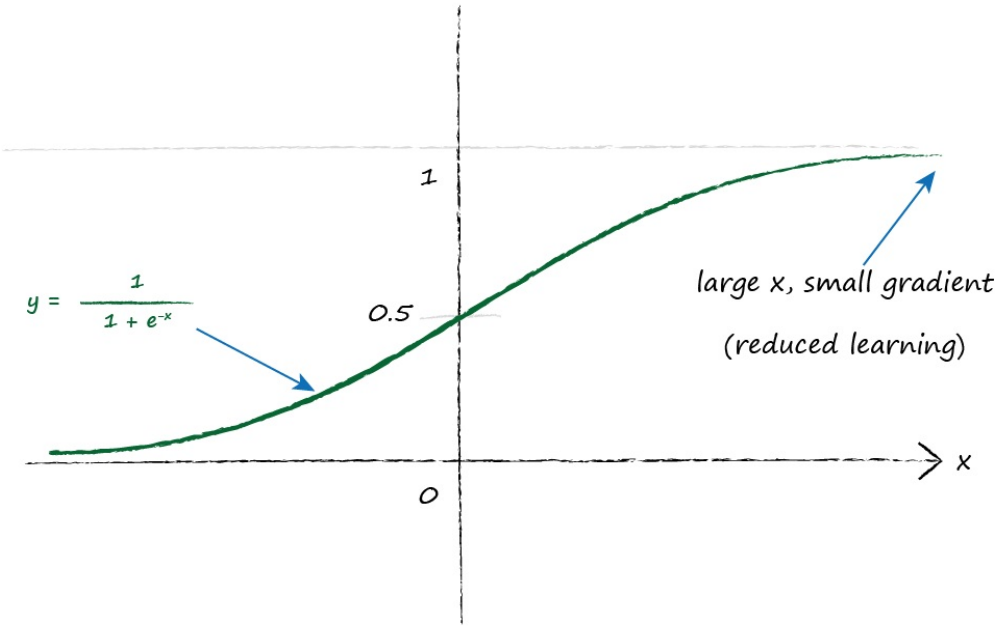
This is quite a small change, but over many hundreds or thousands of iterations the weights will eventually settle down to a configuration so that the well trained neural network produces outputs that reflect the training examples.

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.

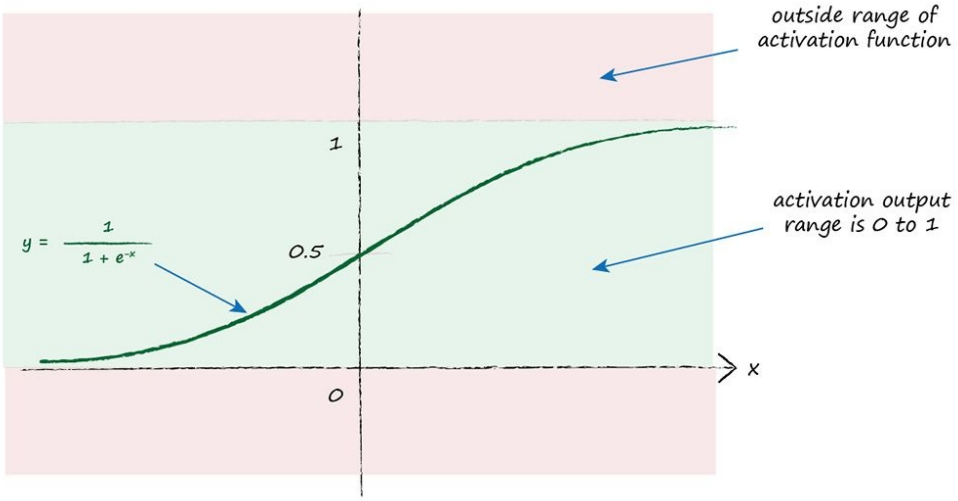
Interestingly, that expression also depends on the incoming signal ($\mathbf{o_j}$) so we shouldn't make it too small either. Very very tiny values can be problematic too because computers can lose accuracy when dealing very very small or very very large numbers.

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.

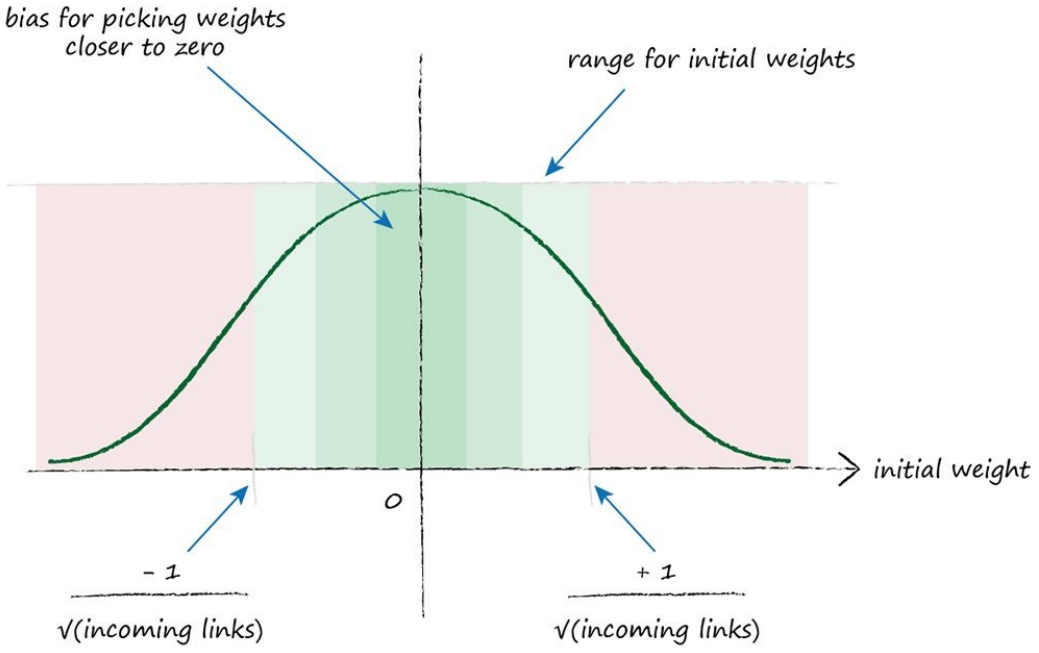
Mathematicians and computer scientists have done the maths to work out a rule of thumb for setting the random initial weights given specific shapes of networks and with specific activation functions. That's a lot of "specifics"! Let's carry on anyway.

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.

Part 2 - DIY with Python

"To really understand something,
you need to make it yourself."

"Start small … then grow"

In this section we'll make our own neural network.

We'll use a computer because, as you know from earlier, there will be many thousands of calculations to do. Computers are good at doing many calculations very quickly, without getting bored or losing accuracy.

We will tell a computer what to do using instructions that it can understand. Computers find human languages like English, French or Spanish hard to understand precisely and without ambiguity. In fact humans have trouble with precision and ambiguity when communicating with each other, so computers have very little hope of doing better!

Python

We'll be using a computer language called **Python**. Python is a good language to start with because it is easy to learn. It's also easy to read and understand someone else's Python instructions. It is also very popular, and used in many different areas, including scientific research, teaching, global scale infrastructures, as well as data analytics and artificial intelligence. Python is increasingly being taught in schools, and the extremely popular Raspberry Pi has made Python accessible to even more people including children and students.

The Appendix includes a guide to setting up a Raspberry Pi Zero to do all the work we've covered in this guide to make your own neural network with Python. A Raspberry Pi Zero is especially inexpensive small computer, and currently costs about £4 or $5. That's not a typo - it really does cost £4.

There is a lot that you can learn about Python, or any other computer language, but here we'll remain focussed on making our own neural network, and only learn just enough Python to achieve this.

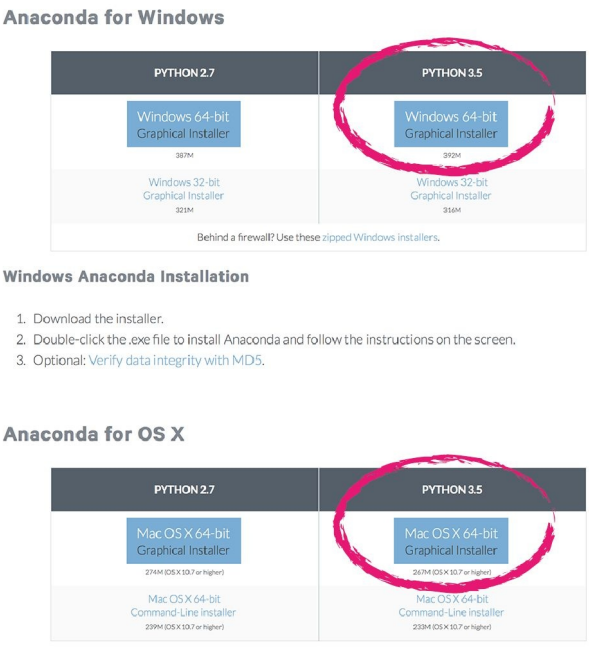
Interactive Python = IPython

Rather than go through the error-prone steps of setting up Python for your

computer, and all the various extensions that help do mathematics and plot images, we're going to use a prepackaged solution, called **IPython**.

IPython contains the Python programming language and several common numerical and data plotting extensions, including the ones we'll need. IPython also has the advantage of presenting interactive notebooks, which behave much like pen and paper notepads, ideal for trying out ideas and seeing the results, and then changing some of your ideas again, easily and without fuss. We avoid worrying about program files, interpreters and libraries, which can distract us from what we're actually trying to do, especially when they don't work as expected.

The ipython.org site gives you some options for where you can get prepackaged IPython. I'm using the **Anaconda** package from www.continuum.io/downloads, shown below.



That site may have changed it's appearance since I took that snapshot, so don't be put off. First go the section for your computer, which might be a Windows computer, or an Apple Mac with OS X, or a Linux computer. Under the section for your computer, make sure you get the **Python 3.5** version and not the 2.7.

Adoption of Python 3 is gathering pace, and is the future. Python 2.7 is well established but we need to move to the future and start using Python 3 whenever

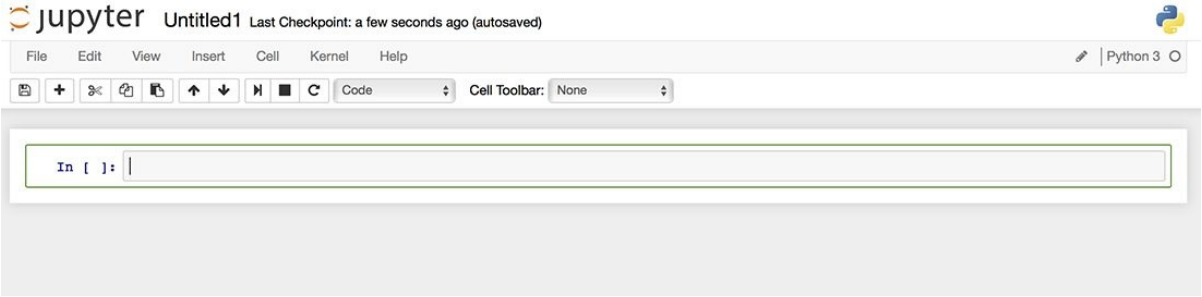
we can, especially for new projects. Most computers have "**64-bit**" brains so make sure you get that version. Only computers roughly more than 10 years old are likely to need the legacy "32-bit" version.

Follow the instructions on that site to get it installed on your computer. Installing IPython should be really easy, because it's designed to be easy, and shouldn't cause any problems.

A Very Gentle Start with Python

We'll assume you now have access to IPython, having followed the instructions for getting it installed.

Notebooks
Once we've fired it up and clicked "New Notebook", we're presented with an empty **notebook** like the following.

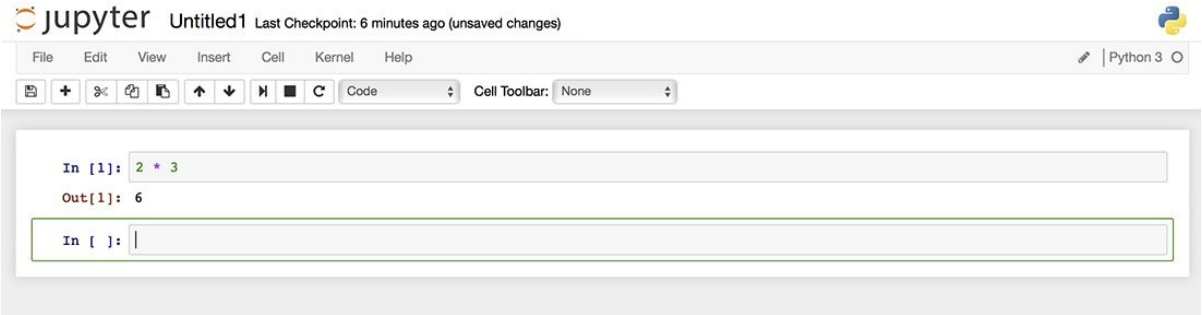


The notebook is interactive, meaning it waits for you to ask it to do something, does it, and then presents back your answer, and waits again for your next instruction or question. It's like a robot butler with a talent for arithmetic that never gets tired.

If you have want to do something that is even mildly complicated, it makes sense to break it down into sections. This makes it easy to organise your thinking, and also find which part of the big project went wrong. For IPython, these sections are called cells. The above IPython notebook has an initial empty cell, and you may be able to see the typing caret blinking, waiting for you to type your instructions into it.

Lets instruct the computer! Let's ask it to multiply two numbers, say 2 times 3. Let's type "2*3" into the cell and click the run cell button that looks like a audio

play button. Don't include the speech marks. The computer should quickly work out what you mean by this, and present the result back to you as follows.



You can see the answer "6" is correctly presented. We've just issued our first instruction to a computer and successfully received a correct result. Our first computer program!

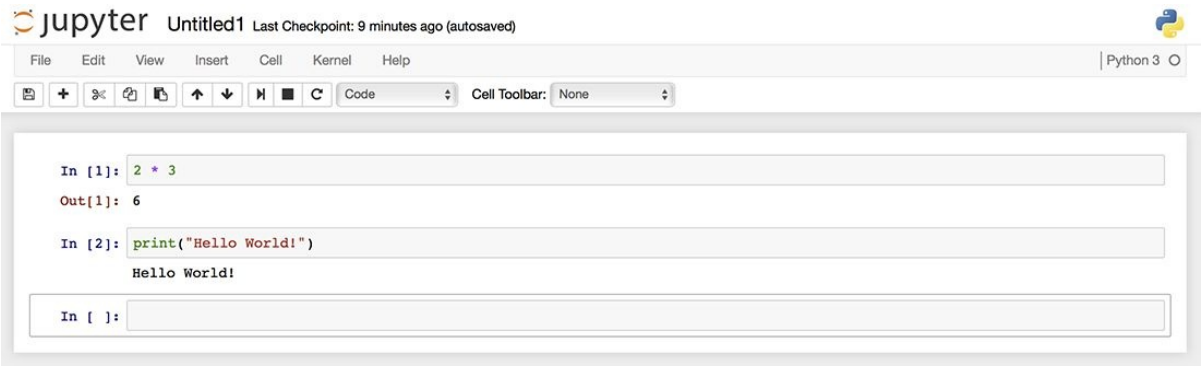
Don't be distracted by IPython labelling your question as "In [1]" and its answer as "Out [1]". That's just it's way of reminding you what you asked (input) and and what it replied with (output). The numbers are the sequence you asked and it responded, useful for keeping track if you find yourself jumping around your notebook adjusting and reissuing your instructions.

Simple Python

We really meant it when we said Python was an easy computer language. In the next ready cell, labelled "In [ ]", type the following code and click play. The word **code** is widely used to refer to instructions written in a computer language. If you find that moving the pointer to click the play button is too cumbersome, like I do, you can use the keyboard shortcut ctrl-enter, instead.

```
print("Hello World!")
```

You should get a response which simply prints the phrase "Hello World!" as follows.

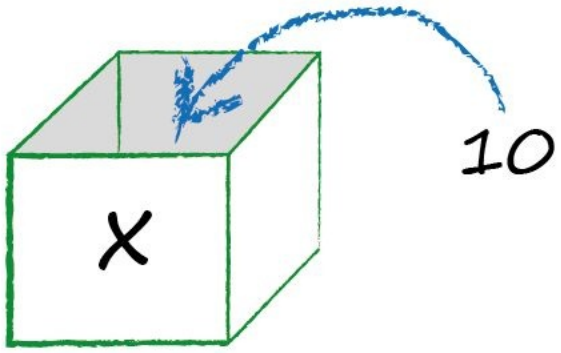


You can see that issuing the second instruction to print “Hello World!” didn’t remove the previous cell with its instruction and output answer. This is useful when slowly building up a solution of several parts.

Now lets see what’s going on with the following code which introduces a key idea. Enter and run it in a new cell. If there is no new empty cell, click the button that looks like a plus sign labelled “Insert Cell Below”.

```
x = 10
print(x)
print(x+5)

y = x+7
print(y)

print(z)
```

The first line “x = 10” looks like a mathematical statement which says x is 10. In Python this means that x is set to 10, that is, the value 10 is placed in an virtual box called x. It’s as simple as the following diagram shows.

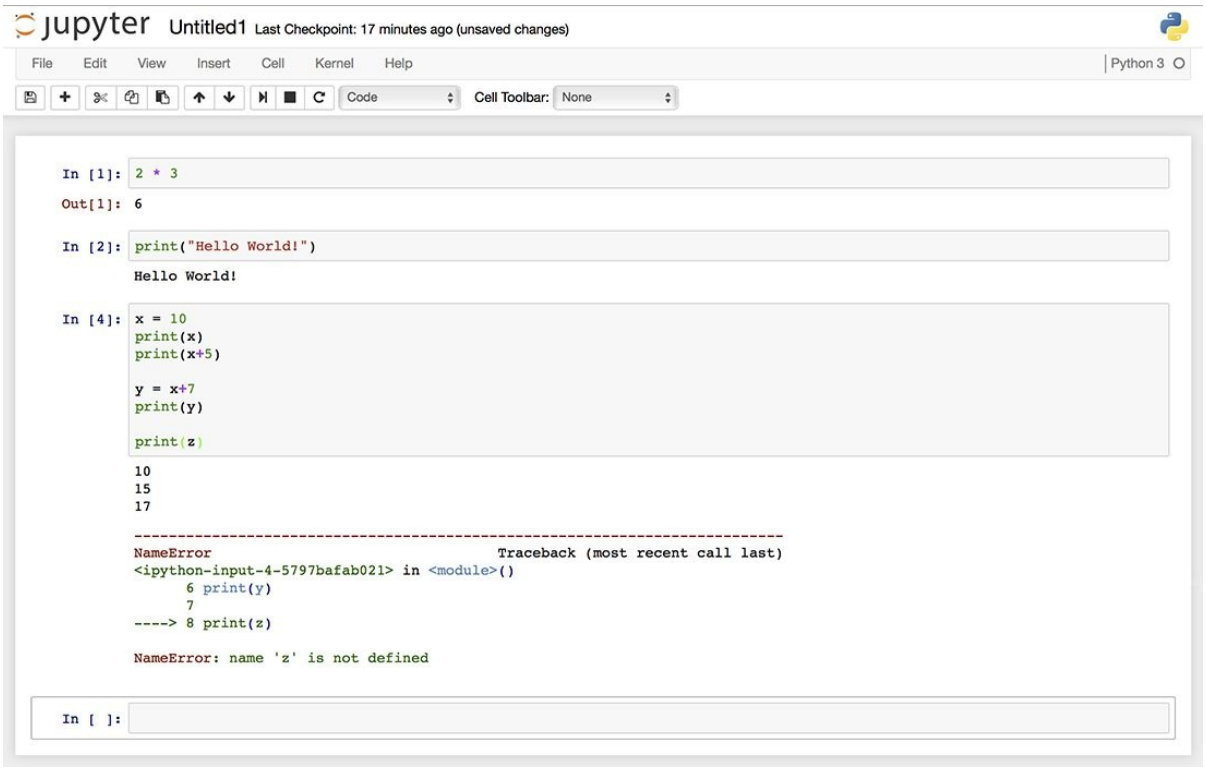


That 10 stays there until further notice. We shouldn't be surprised by the "print(x)" because we used the print instruction before. It should print the value of x, which is "10". Why doesn't it just print the letter "x"? Because Python is always eager to evaluate whatever it can, and x can be evaluated to the value 10 so it prints that. The next line "print (x+5)" evaluates x+5, which is 10+5 or 15, so we expect it to print "15".

The next bit "y = x+7" again shouldn't be difficult you work out if we follow this idea that Python evaluates whatever it can. We've told it to assign a value to a new box labelled y, but what value? The expression is x+7, which is 10+7, or 17. So y holds the value 17, and the next line should print it.

What happens with the line "print(z)" when we haven't assigned a value to z like we have with x and y? We get an error message which is polite and tells us about the error of our ways, trying to be helpful as possible so we can fix it. I have to say, most computer languages have error messages which try to be helpful but don't always succeed.

The following shows the results of the above code, including the helpful polite error message, "name z is not defined".



These boxes with labels like x and y, which hold values like 10 and 17, are called **variables**. Variables in computer languages are used to make a set of instructions generic, just like mathematicians use expressions like “x” and “y” to make general statements.

Automating Work

Computers are great for doing similar tasks many times - they don’t mind and they’re very quick compared to humans with calculators!

Let’s see if we can get a computer to print the first ten squared numbers, starting with 0 squared, 1 squared, then 2 squared and so on. We expect to see an output of something like 0, 1, 4, 9, 16, 25, and so on.

We could just do the calculation ourselves, and have a set of instructions like “print (0)”, “print (1)”, “print (4)”, and so on. This would work but we would have failed to get the computer to do the calculation for us. More than that, we would have missed the opportunity to have a generic set of instruction to print the squares of numbers up to any specified value. To do this we need to pick up a few more new ideas, so we’ll take it gently.

Issue the following code into the next ready cell and run it.

```
list( range(10) )
```

You should get a list of ten numbers, from 0 up to 9. This is great because we got the computer to do the work to create the list, we didn’t have to do it ourselves. We are the master and the computer is our servant!

```
In [8]: list( range(10) )
Out[8]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You may have been surprised that the list was from 0 to 9, and not from 1 to 10. This is because many computer related things start with 0 and not 1. It’s tripped me up many times when I assumed a computer list started with 1 and not 0. Creating ordered lists are useful to keep count when performing calculations, or indeed applying iterative functions, many times.

You may have noticed we missed out the “print” keyword, which we used when

we printed the phrase “Hello World!”, but again didn’t when we evaluated 2*3. Using the “print” keyword can be optional when we’re working with Python in an interactive way because it knows we want to see the result of the instructions we issued.

A very common way to get computers to do things repeatedly is by using code structures called **loops**. The word loop does give you the right impression of something going round, and round potentially endlessly. Rather than define a loop, it’s easiest to see a simple one. Enter and run the following code in a new cell.

```
for n in range(10):
        print(n)
        pass
print(“done”)
```

There are three new things here so let’s go through them. The first line has the “range(10)” that we saw before. This creates a list of numbers from 0 to 9, as we saw before.

The “for n in” is the bit that creates a loop, and in this case it does something for every number in the list, and keeps count by assigning the current value to the variable n. We saw variables earlier and this is just like assigning n=0 during the first pass of the loop, then n=1, then n=2, until n=9 which is the last item in the list.

The next line “print(n)” shouldn’t surprise us by simply printing the value of n. We expect all the numbers in the list to be printed. But notice the indent before “print(n)”. This is important in Python as indents are used meaningfully to show which instructions are subservient to others, in this case the loop created by “for n in ...“. The “pass” instruction signals the end of the loop, and the next line is back at normal indentation and not part of the loop. This means we only expect “done” to be printed once, and not ten times. The following shows the output, and it is as we expected.

```
Out[8]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [12]: for n in range(10):
             print(n)
             pass
         print("done")

         0
         1
         2
         3
         4
         5
         6
         7
         8
         9
         done
```

It should be clear now that we can print the squares by printing “n*n”. In fact we can make the output more helpful by printing phrases like “The square of 3 is 9”. The following code shows this change to the print instruction repeated inside the loop. Note how the variables are not inside quotes and are therefore evaluated.

```
for n in range(10):
    print("The square of", n, "is", n*n)
    pass
print("done")
```

The result is shown as follows.

```
In [13]: for n in range(10):
             print("The square of", n, "is", n*n)
             pass
         print("done")

         The square of 0 is 0
         The square of 1 is 1
         The square of 2 is 4
         The square of 3 is 9
         The square of 4 is 16
         The square of 5 is 25
         The square of 6 is 36
         The square of 7 is 49
         The square of 8 is 64
         The square of 9 is 81
         done
```

This is already quite powerful! We can get the computer to potentially do a lot of work very quickly with just a very short set of instructions. We could easily make the number of loop iterations much larger by using range(50) or even range(1000) if we wanted. Try it!

Comments

Before we discover more wild and wonderful Python commands, have a look at the following simple code.

```
# the following prints out the cube of 2
print(2**3)
```

The first line begins with a hash symbol #. Python ignores any lines beginning with a hash. Rather than being useless, we can use such lines to place helpful comments into the code to make it clearer for other readers, or even ourselves if we came back to the code at a later time.

Trust me, you'll be thankful you commented your code, especially the more complex or less obvious bits of code. The number of times I've tried to decode my own code, asking myself "what was I thinking …"

Functions

We spent a lot of time earlier in part 1 of the guide working with mathematical functions. We thought of these as machines which take input, do some work, and pop out the result. And those functions stood in their own right, and could be used again and again.

Many computer languages, Python included, make it easy to create reusable computer instructions. Like mathematical functions, these reusable snippets of code stand on their own if you define them sufficiently well, and allow you to write shorter more elegant code. Why shorter code? Because invoking a function by its name many times is better than writing out all the function code many times.

And what do we mean by "sufficiently well defined"? It means being clear about what kinds of input a function expects, and what kind of output it produces. Some functions will only take numbers as input, so you can't supply them with a word made up of letters.

Again, the best way to understand this simple idea of a **function** is to see a simple one and play with it. Enter the following code and run it.

```
# function that takes 2 numbers as input
# and outputs their average
def avg(x,y):
    print("first input is", x)
    print("second input is", y)
    a = (x + y) / 2.0
```

```
    print("average is", a)
    return a
```

Let's talk about what we've done here. The first two lines starting with # are ignored by Python but for us can be used as comments for future readers. The next bit "def avg(**x,y**)" tells Python we are about define a new reusable function. That's the "def" keyword. The "avg" bit is the name we've given it. It could have been called "banana" or "pluto" but it makes sense to use names that remind us what the function actually does. The bits in brackets (**x,y**) tells Python that this function takes two inputs, to be called **x** and **y** inside the forthcoming definition of the function. Some computer languages make you say what kind of objects these are, but Python doesn't do this, it just complains politely later when you try to abuse a variable, like trying to use a word as if it was a number, or other such insanity.

Now that we've signalled to Python that we're about to define a function, we need to actually tell it what the function is to do. This definition of the function is indented, as shown in the code above. Some languages use lots of brackets to make it clear which instructions belong to which parts of a program, but the Python designers felt that lots of brackets weren't easy on the eye, and that indentation made understanding the structure of a program instantly visual and easier. Opinions are divided because people get caught out by such indentation, but I love it! It's one of the best human-friendly ideas to come out of the sometimes geeky world of computer programming!

The definition of the avg(**x,y**) function is easy to understand as it uses only things we've seen already. It prints out the first and second numbers which the function gets when it is invoked. Printing these out isn't necessary to work out the average at all, but we've done it to make it really clear what is happening inside the function. The next bit calculates (**x**+**y**)/2.0 and assigns the value to the variable named **a**. We again print the average just to help us see what's going on in the code. The last statement says "return a". This is is the end of the function and tells Python what to throw out as the functions output, just like machines we considered earlier.

When we ran this code, it didn't seem to do anything. There were no numbers produced. That's because we only defined the function, but haven't used it yet. What has actually happened is that Python has noted this function and will keep it ready for when we want to use it.

In the next cell enter “avg(2,4)” to invoke this function with the inputs 2 and 4. By the way, invoking a function is called **calling a function** in the world of computer programming. The output should be what we expect, with the function printing a statement about the two input values and the average it calculated. You’ll also see the answer on it’s own, because calling the function in an interactive Python sessions prints out the returned value. The following shows the function definition and the results of calling it with avg(2,4) and also bigger values (200, 301). Have a play and experiment with your own inputs.

```
In [20]: # function that takes 2 numbers as input
         # and outputs their average
         def avg(x,y):
             print("first input is", x)
             print("second input is", y)
             a = (x + y) / 2.0
             print("average is", a)
             return a

In [21]: avg(2,4)

         first input is 2
         second input is 4
         average is 3.0

Out[21]: 3.0

In [23]: avg(200,301)

         first input is 200
         second input is 301
         average is 250.5

Out[23]: 250.5
```

You may have noticed that the function code which calculates the average divides the sum of the two inputs by 2.0 and not just 2. Why is this? Well this is a peculiarity of Python which I don’t like. If we used just “2” the result would be rounded down to the nearest whole number, because Python considers just “2” as an integer. This would be fine for avg(2,4) because 6/2 is 3, a whole number. But for avg(200,301) the average is 501/2 which should be 250.5 but would be rounded down to 250. This is all just very silly I think, but worth thinking about if your own code isn’t behaving quite right. Dividing by “2.0” tells Python we really want to stick with numbers that can have fractional parts, and we don’t want it to round down to whole numbers.

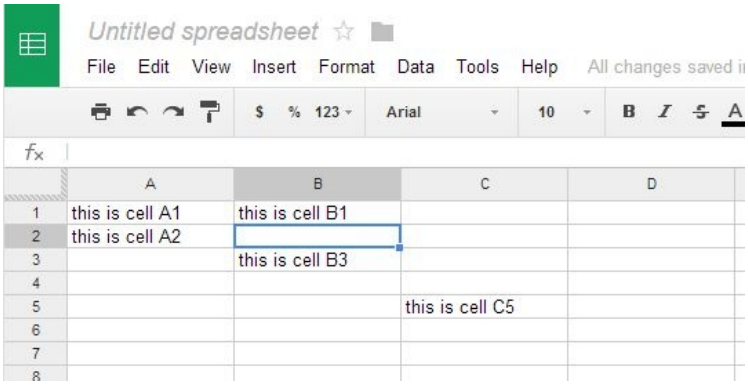
Let’s take a step back and congratulate ourselves. We’ve defined a reusable function, one of the most important and powerful elements of both mathematics and in computer programming.

We’ll use reusable functions when we code our own neural network. For

example, it makes sense to make a reusable function that does the sigmoid activation function calculation so we can call on it many times.

Arrays

Arrays are just tables of values, and they come in really handy. Like tables, you refer to particular cells according to the row and column number. If you think of spreadsheets, you'll know that cells are referred to in this way, B1 or C5 for example, and the values in those cells can be used in calculations, C3+D7 for example.



When we get to coding our neural network, we'll use arrays to represent the matrices of input signals, weights and output signals too. And not just those, we'll use them to represent the signals inside the neural networks as they feed forward, and also the errors as they propagate backwards. So let's get familiar with them. Enter and run the following code.

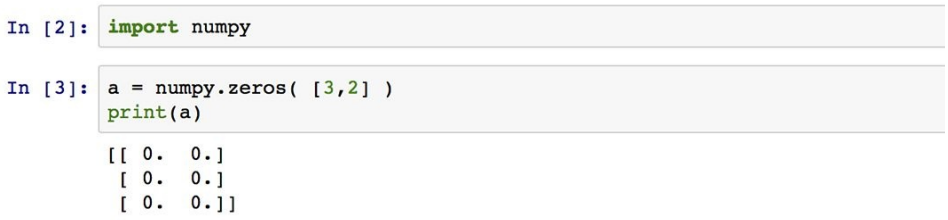
```
import numpy
```

What does this do? The import command tells Python to bring additional powers from elsewhere, to add new tools to it's belt. Sometimes these additional tools are part of Python but aren't made immediately ready to use, to keep Python lean and mean, and only carry extra stuff if you intend to use it. Often these additional tools not core parts of Python, but are created by others as useful extras, contributed for everyone to use. Here we've imported an additional set of tools packaged into a module named **numpy**. Numpy is really popular, and contains some useful stuff, like arrays and an ability to do calculations with them.

In the next cell, the following code.

```
a = numpy.zeros( [3,2] )
print(a)
```

This uses the imported numpy module to create an array of shape 3 by 2, with all the cells set to the value zero and assigns the whole thing to a variable named **a**. We then print **a**. We can see this array full of zeros in what looks like a table with 3 rows and 2 columns.



Now let's modify the contents of this array and change some of those zeros to other values. The following code shows how you can refer to specific cells to overwrite them with new values. It's just like referring to spreadsheet cells or street map grid references.

```
a[0,0] = 1
a[0,1] = 2
a[1,0] = 9
a[2,1] = 12
print(a)
```

The first line updates the cell at row zero and column zero with the value 1, overwriting whatever was there before. The other lines are similar updates, with a final printout with "print(a)". The following shows us what the array looks like after these changes.

```
In [2]: import numpy

In [3]: a = numpy.zeros( [3,2] )
        print(a)

        [[ 0.  0.]
         [ 0.  0.]
         [ 0.  0.]]

In [4]: a[0,0] = 1
        a[0,1] = 2
        a[1,0] = 9
        a[2,1] = 12
        print(a)

        [[  1.   2.]
         [  9.   0.]
         [  0.  12.]]
```

Now that we know how to set the value of cells in an array, how do we look them up without printing out the entire array? We've been doing it already. We simply use the expressions like **a**[1,2] or **a**[2,1] to refer to the content of these cells which we can print or assign to other variables. The code shows us doing just this.

```
print(a[0,1])
v = a[1,0]
print(v)
```

You can see from the output that the first print instruction produced the value 2.0 which is what's inside the cell at [0,1]. Next the value inside **a**[1,0] is assigned to the variable **v** and this variable is printed. We get the expected 9.0 printed out.

```
In [5]: print(a[0,1])
        v = a[1,0]
        print(v)

        2.0
        9.0
```

The column and row numbering starts from 0 and not 1. The top left is at [0,0] not [1,1]. This also means that the bottom right is at [2,1] not [3,2]. This catches me out sometimes because I keep forgetting that many things in the computer world begin with 0 and not 1.If we tried to refer to **a**[3,2] we'd get an error message telling us we were trying to locate a cell which didn't exist. We'd get the same if we mixed up our columns and rows. Let's try accessing **a**[0,2] which doesn't exist just to see what error message is reported.

```
In [6]: # trying to look up an array element that doesn't exist
        a[0,2]

        ---------------------------------------------------------------------------
        IndexError                                Traceback (most recent call last)
        <ipython-input-6-489d1c44975f> in <module>()
              1 # trying to look up an array element that doesn't exist
        ----> 2 a[0,2]

        IndexError: index 2 is out of bounds for axis 1 with size 2
```

Arrays, or matrices, will be useful for neural networks because we can simplify the instructions to do the many many calculations for feeding signals forward and errors backwards through a network. We saw this in part 1 of this guide.

Plotting arrays

Just like large tables or lists of numbers, looking at large arrays isn't that insightful. Visualising them helps us quickly get an idea of the general meaning. One way of plotting 2-dimensional arrays of numbers is think of them as flat 2-dimensional surfaces, coloured according to the value at each cell in the array. You can choose how you turn a value inside a cell into a colour. You might choose to simply turn the value into a colour according to a colour scale, or you might colour everything white except values above a certain threshold which would be black.

Let's try plotting the small 3 by 2 array we created above.

Before we can do this, we need to extend Python's abilities to plot graphics. We do this by **importing** additional Python code that others have written. You can think of this as borrowing food recipe books from your friend to add to your own bookshelf, so that your bookshelf now has extra content enabling you to prepare more kinds of dishes than you could before.

The following shows how we import graphics plotting capability.

```
import matplotlib.pyplot
```

The "matplotlib.pyplot" is the name of the new "recipe book" that we're borrowing. You might come across phrases like "import a module" or "import a library". These are just the names given to the additional Python code you're importing. If you get into Python you'll often import additional capabilities to make your life easier by reusing other's useful code. You might even create your own useful code to share with others!

One more thing, we need to be firm with IPython about plotting any graphics in the notebook, and not try to plot it in a separate external window. We issue this explicit order as follows:

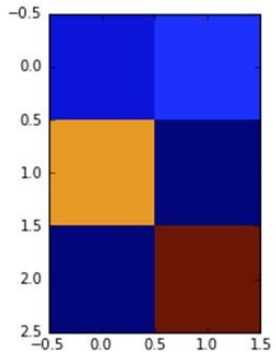
```
%matplotlib inline
```

We're ready to plot that array now. Enter and run the following code.

```
matplotlib.pyplot.imshow(a, interpolation="nearest")
```

The instruction to create a plot is imshow(), and the first parameter is the array we want to plot. That last bit "interpolation" is there to tell Python not to try to blend the colours to make the plot look smoother, which it does by default, trying to helpful. Let's look at the output.

```
In [8]: import matplotlib.pyplot
        %matplotlib inline

In [9]: matplotlib.pyplot.imshow(a, interpolation="nearest")
Out[9]: <matplotlib.image.AxesImage at 0x108917710>
```



Exciting! Our first plot shows the 3 by 2 sized array as colours. You can see that the array cells which have the same value also have the same colour. Later we'll be using this very same imshow() instruction to visualise an array of values that we feed our neural network.

The IPython package has a very rich set of tools for visualising data. You should explore them to get a feel for the wide range of plots possible, and even try some of them. Even the imshow() instruction has many options for plotting for us to explore, such as using different colour palettes.

Objects

We're going to look at one more Python idea called **objects**. Objects are similar to the reusable functions because we define them once and use them many times. But objects can do a lot more than simple functions can.

The easiest way to understand objects is to see them in action, rather than talk loads about an abstract concept. Have a look at the following code.

```
# class for a dog object
class Dog:

    # dogs can bark()
    def bark(self):
        print("woof!")
        pass

    pass
```

Let's start with what we're familiar with first. You can see there is a function called bark() inside that code. It's not difficult to see that if we called the function, we'd get "woof!" printed out. That's easy enough.

Let's look around that familiar function definition now. You can see the class keyword, and a name "Dog" and a structure that looks like a function too. You can see the parallels with function definitions which also have a name. The difference is that with functions we use the "def" keyword to define them, but with object definitions we use the keyword "class".

Before we dive in and discuss what a **class** is, compared to an object, again have a look at some real but simple code that brings these abstract ideas to life.

```
sizzles = Dog()
sizzles.bark()
```

You can see the first line creates a variable called **sizzles**, and it seems to come from what looks like a function call. In fact that Dog() is a special function that creates an instance of the Dog class. We can see now how to create things from class definitions. These things are called **objects**. We've created an object called **sizzles** from the Dog class definition - we can consider this object to be a dog!

The next line calls the bark() function on the **sizzles** object. This is half familiar because we've seen functions already. What's less familiar is that we're calling the bark() function as if it was a part of the **sizzles** object. That's because bark() is a function that all objects created from the Dog class have. It's there to see in the definition of the Dog class.

Let's say all that in simple terms. We created **sizzles**, a kind of Dog. That **sizzles** is an object, created in the shape of a Dog class. Objects are instances of a class.

The following shows what we have done so far, and also confirms that **sizzles**.bark() does indeed output a "woof!"

```
In [7]: # class for a dog object
        class Dog:

            # dogs can bark()
            def bark(self):
                print("woof!")
                pass

            pass

In [8]: sizzles = Dog()

In [9]: sizzles.bark()
        woof!
```

You might have spotted the "self" in the definition of the function as bark(self). This may seem odd, and it does to me. As much as I like Python, I don't think it is perfect. The reason that "self" is there is so that when Python creates the function, it assigns it to the right object. I think this should be obvious because that bark() is inside the class definition so Python should know which objects to connect it to, but that's just my opinion.

Let's see objects and classes being used more usefully. Have a look at the following code.

```
sizzles = Dog()
mutley = Dog()

sizzles.bark()
```

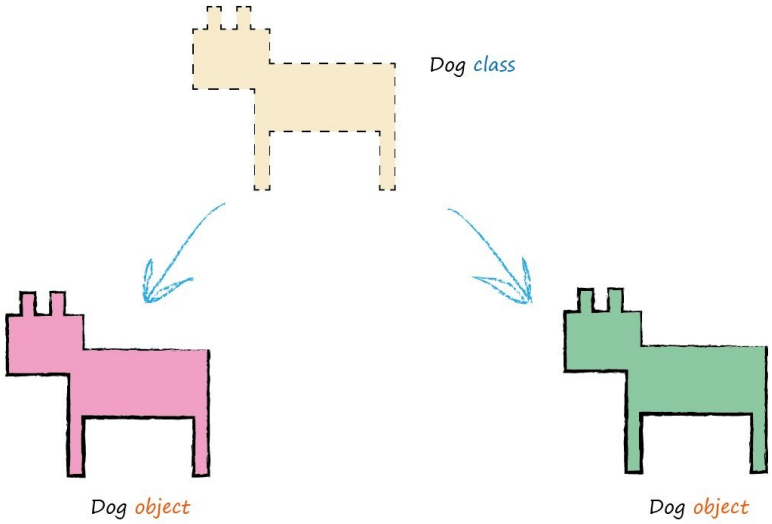
```
mutley.bark()
```

Let’s run it and see what happens.

```
In [4]: sizzles = Dog()
        mutley = Dog()

        sizzles.bark()
        mutley.bark()

        woof!
        woof!

In [ ]:
```

This is interesting! We are creating two objects called sizzles and mutley. The important thing to realise is that they are both created from the same Dog() class definition. This is powerful! We define what the objects should look like and how they should behave, and then we create real instances of them.

That is the difference between **classes** and **objects**, one is a definition and one is a real instance of that definition. A class is a cake recipe in a book, an object is a cake made with that recipe. The following shows visually how objects are made from a class recipe.



What use are these objects made from a class? Why go to the trouble? It would

have been simpler to just print the word “woof!” without all that extra code.

Well, you can see how it might be useful to have many objects all of the same kind, all made from the same template. It saves having to create each one separately in full. But the real benefit comes from objects having data and functions all wrapped up neatly inside them. The benefit is to us human coders. It helps us more easily understand more complicated problems if bits of code are organised around objects to which they naturally belong. Dogs bark. Buttons click. Speakers emit sound. Printers print, or complain they’re out of paper. In many computer systems, buttons, speakers and printers are indeed represented as objects, whose functions you invoke.

You’ll sometimes see these object functions called **methods**. We’ve already done this above, we added a bark() function to the Dog class and both the sizzles and mutley objects made from this class have a bark() method. You saw them both bark in the example!

Neural networks take some input, do some calculations, and produce an output. We also know they can be trained. You can see that these actions, training and producing an answer, are natural functions of a neural network. That is, functions of a neural network object. You’ll also remember that neural networks have data inside them that naturally belongs there - the link weights. That’s why we’ll build our neural network as an object.

For completeness, let’s see how we add data variables to a class, and some methods to view and change this data. Have a look at the following fresh class definition of a Dog class. There’s a few new things going on here so we’ll look at them one at a time.

```
# class for a dog object
class Dog:

    # initialisation method with internal data
    def __init__(self, petname, temp):
        self.name = petname;
        self.temperature = temp;

    # get status
    def status(self):
```

```
        print("dog name is ", self.name)
        print("dog temperature is ", self.temperature)
        pass

    # set temperature
    def setTemperature(self,temp):
        self.temperature = temp;
        pass

    # dogs can bark()
    def bark(self):
        print("woof!")
        pass

    pass
```

The first thing to notice is we have added three new functions to this Dog class. We already had the bark() function, now have new ones called __init__(), status() and setTemperature(). Adding new functions is easy enough to understand. We could have added one called sneeze() to go with bark() if we wanted.

But these new functions seem to have variable names inside the function names. That setTemperature is actually setTemperature(self, **temp**). The funny named __init__ is actually __init__(self, **petname**, **temp**). What are these extra bits inside the brackets? They are the variables the function expects when it is called - called **parameters**. Remember that averaging function avg(**x,y**) we saw earlier? The definition of avg() made clear it expected 2 numbers. So the __init__() function needs a **petname** and a **temp**, and the setTemperature() function needs just a **temp**.

Now let's look inside these new functions. Let's look at that oddly named __init__() first. Why is it given such a weird name? The name is special, and Python will call a function called __init__() when an object is first created. That's really handy for doing any work preparing an object before we actually use it. So what do we do in this magic initialisation function? We seem to only create 2 new variables, called self.**name** and self.**temperature**. You can see their values from the variables **petname** and **temp** passed to the function. The "self." part means that the variables are part of this object itself - it's own "self".

That is, they belong only to this object, and are independent of another Dog object or general variables in Python. We don't want to confuse this dog's name with that of another dog! If this all sounds complicated, don't worry, it'll be easy to see when we actually run a real example.

Next is the status() function, which is really simple. It doesn't take any parameters, and simply prints out the Dog object's name and temperature variables.

Finally the setTemperature() function does take a parameter. You can see that it sets the self.**temperature** to the parameter temp supplied when this function is called. This means you can change the object's temperature at any time after you created the object. You can do this as many times as you like.

We've avoiding talking about why all these functions, including that bark() function have a "self" as the first parameter. It is a peculiarity of Python, which I find a bit irksome, but that's the way Python has evolved. What it does is make clear to Python that the function you're about to define belongs to the object, referred to as "self". You would have thought it was obvious because we're writing the function inside the class. You won't be surprised that this has caused debates amongst even expert Python programmers, so you're in good company in being puzzled.

Let's see all this in action to make it really clear. The following shows the new Dog class defined with these new functions, and a new Dog object called **lassie** being created with parameters setting out it's name as "Lassie" and it's initial temperature as 37.

In [18]:
```
# class for a dog object
class Dog:

    # initialisation method with internal data
    def __init__(self, petname, temp):
        self.name = petname;
        self.temperature = temp;

    # get status
    def status(self):
        print("dog name is ", self.name)
        print("dog temperature is ", self.temperature)
        pass

    # set temperature
    def setTemperature(self,temp):
        self.temperature = temp;
        pass

    # dogs can bark()
    def bark(self):
        print("woof!")
        pass

    pass
```

In [19]:
```
# create a new dog object from the Dog class
lassie = Dog("Lassie", 37)
```

In [20]:
```
lassie.status()
```

```
dog name is  Lassie
dog temperature is  37
```

You can see how calling the status() function on this **lassie** Dog object prints out the dog's name and it's current temperature. That temperature hasn't changed since lassie was created.

Let's now change the temperature and check the it really has changed inside the object by asking for another update:

```
lassie.setTemperature(40)
lassie.status()
```

The following shows the results.

```
In [19]: # create a new dog object from the Dog class
         lassie = Dog("Lassie", 37)

In [20]: lassie.status()

         dog name is  Lassie
         dog temperature is  37

In [22]: lassie.setTemperature(40)

In [23]: lassie.status()

         dog name is  Lassie
         dog temperature is  40
```

You can see that calling setTemperature(40) on the lassie object did indeed change the object's internal record of the temperature.

We should be really pleased because we've learned quite a lot about objects, which some consider an advanced topic, and it wasn't that hard at all!

We've learned enough Python to begin making our neural network.

Neural Network with Python

We'll now start the journey to make our own neural network with the Python we've just learned. We'll progress along this journey is short easy to tackle steps, and build up a Python program bit by bit.

Starting small, then growing, is a wise approach to building computer code of even moderate complexity.

After the work we've just done, it seems really natural to start to build the skeleton of a neural network class. Let's jump right in!

The Skeleton Code
Let's sketch out what a neural network class should look like. We know it should have at least three functions:

- initialisation - to set the number of input, hidden and output nodes

- train - refine the weights after being given a training set example to learn from
- query - give an answer from the output nodes after being given an input

These might not be perfectly defined right now, and there may be more functions needed, but for now let's use these to make a start.

The code taking shape would be something like the following:

```
# neural network class definition
class neuralNetwork:

    # initialise the neural network
    def __init__():
        pass

    # train the neural network
    def train():
        pass

    # query the neural network
    def query():
        pass
```

That's not a bad start at all. In fact that's a solid framework on which to flesh out the working of the neural network in detail.

Initialising the Network

Let's begin with the initialisation. We know we need to set the number of input, hidden and output layer nodes. That defines the shape and size of the neural network. Rather than set these in stone, we'll let them be set when a new neural network object is created by using parameters. That way we retain the choice to create new neural networks of different sizes with ease.

There's an important point underneath what we've just decided. Good programmers, computer scientists and mathematicians, try to create general code rather than specific code whenever they can. It is a good habit, because it forces us to think about solving problems in a deeper more widely applicable way. If we do this, then our solutions can be applied to different scenarios. What this means here is that we will try to develop code for a neural network which tries to keep as many useful options open, and assumptions to a minimum, so that the

code can easily be used for different needs. We want the same class to be able to create a small neural network as well as a very large one - simply by passing the desired size as parameters.

Don't forget the learning rate too. That's a useful parameter to set when we create a new neural network. So let's see what the __init__() function might look like:

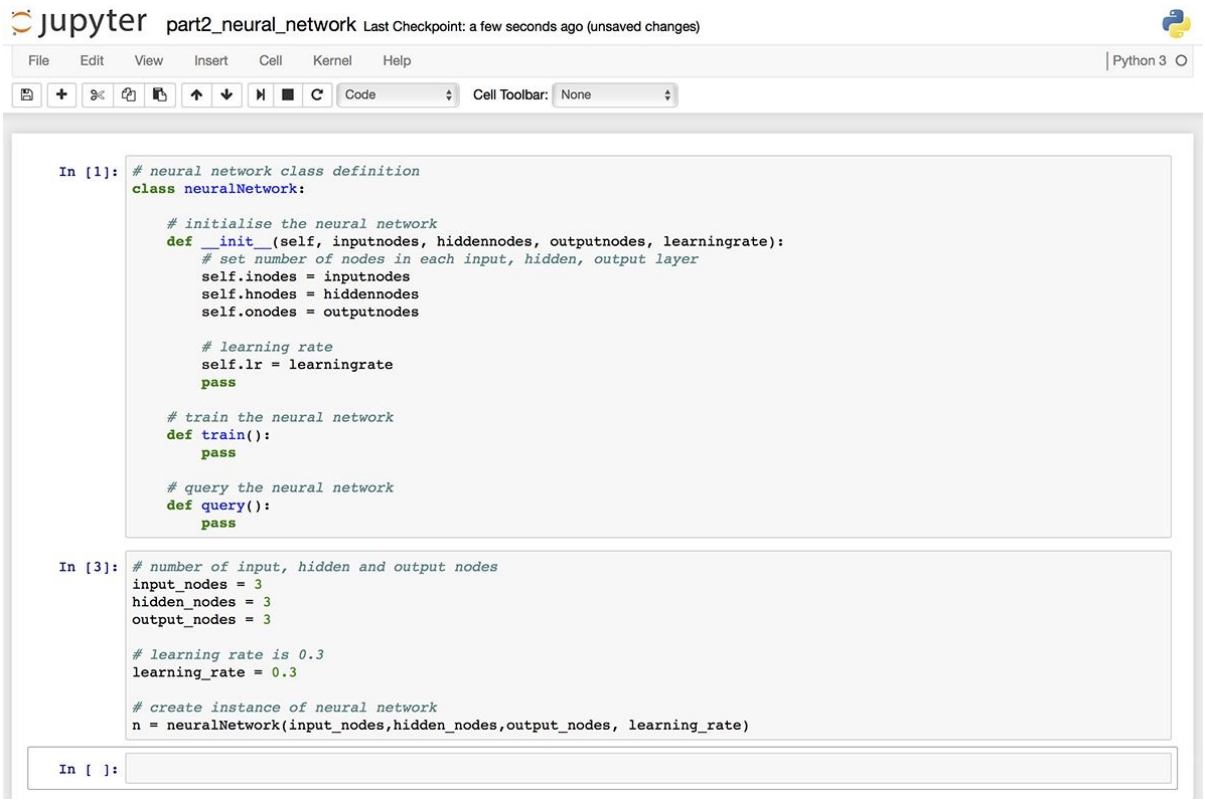
```
    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # learning rate
        self.lr = learningrate
        pass
```

Let's add that to our class definition of a neural network, and try creating a small neural network object with 3 nodes in each layer, and a learning rate of 0.5.

```
# number of input, hidden and output nodes
input_nodes = 3
hidden_nodes = 3
output_nodes = 3

# learning rate is 0.3
learning_rate = 0.3

# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
learning_rate)
```

That would give us an object, sure, but it wouldn't be very useful yet as we haven't coded any functions that do useful work yet. That's ok, it is a good technique to start small and grow code, finding and fixing problems along the way.

Just to make sure we've not got lost, the following shows the IPython notebook at this stage, with the neural network class definition and the code to create an object.



What do we do next? Well we've told the neural network object how many input, hidden and output layer nodes we want, but nothing has really been done about it.

Weights - The Heart of the Network

So the next step is to create the network of nodes and links. The most important part of the network is the **link weights**. They're used to calculate the signal being fed forward, the error as it's propagated backwards, and it is the link weights themselves that are refined in an attempt to to improve the network.

We saw earlier that the weights can be concisely expressed as a matrix. So we can create:

- A matrix for the weights for links between the input and hidden layers, $\mathbf{W_{input\_hidden}}$, of size (**hidden_nodes** by **input_nodes)**.
- And another matrix for the links between the hidden and output layers, $\mathbf{W_{hidden\_output}}$, of size (**output_nodes** by **hidden_nodes)**.

Remember the convention earlier to see why the first matrix is of size (**hidden_nodes** by **input_nodes)** and not the other way around (**input_nodes** by

hidden_node).

Remember from part 1 of this guide that the initial values of the link weights should be small and random. The following numpy function generates an array of values selected randomly between 0 and 1, where the size is (rows by columns).

```
numpy.random.rand(rows, columns)
```

All good programmers use internet search engines to find online documentation on how to use cool Python functions, or even to find useful functions they didn't know existed. Google is particularly good for finding stuff about programming. This numpy.random.rand() function is described here, for example.

If we're going to use the numpy extensions, we need to import the library at the top of our code. Try this function, and confirm to yourself it works. The following shows it working for a (3 by 3) numpy array. You can see that each value in the array is random and between 0 and 1.

```
In [1]: import numpy

In [3]: numpy.random.rand(3, 3)

Out[3]: array([[ 0.8133122 ,  0.49193566,  0.14790496],
               [ 0.75997346,  0.15676617,  0.27449845],
               [ 0.03287221,  0.01884548,  0.17282894]])
```

We can do better because we've ignored the fact that the weights could legitimately be negative not just positive. The range could be between -1.0 and +1.0. For simplicity, we'll simply subtract 0.5 from each of the above values to in effect have a range between -0.5 to +0.5. The following shows this neat trick working, and you can see some random values below zero.

```
In [5]: numpy.random.rand(3, 3) - 0.5

Out[5]: array([[ 0.143827  , -0.13728512,  0.24625022],
               [-0.41129188,  0.24551424, -0.43500754],
               [ 0.3188901 ,  0.06173198,  0.18406137]])
```

We're ready to create the initial weight matrices in our Python program. These weights are an intrinsic part of a neural network, and live with the neural network for all its life, not a temporary set of data that vanishes once a function is called. That means it needs to be part of the initialisation too, and accessible from other functions like the training and the querying.

The following code, with comments included, creates the two link weight matrices using the self.**inodes**, self.**hnodes** and self.**onodes** to set the right size for both of them.

```
# link weight matrices, wih and who
# weights inside the arrays are w_i_j, where link is from node i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = (numpy.random.rand(self.hnodes, self.inodes) - 0.5)
self.who = (numpy.random.rand(self.onodes, self.hnodes) - 0.5)
```

Great work! We've implemented the very heart of a neural network, its link weight matrices!

Optional: More Sophisticated Weights
This bit is optional as it is just a simple but popular refinement to initialising weights.

If you followed the earlier discussion in part 1 of this guide about preparing data and initialising weights at the end of the first part of this guide, you will have seen that some prefer a slightly more sophisticated approach to creating the initial random weights. They sample the weights from a normal probability distribution centred around zero and with a standard deviation that is related to the number of incoming links into a node, 1/√(number of incoming links).

This is easy to do with the help of the numpy library. Again Google is really helpful in finding the right documentation. The numpy.random.normal() function described here, helps us sample a normal distribution. The parameters are the centre of the distribution, the standard deviation and the size of a numpy array if we want a matrix of random numbers instead of just a single number.

Our updated code to initialise the weights will look like this:

```
self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5), (self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5), (self.onodes, self.hnodes))
```

You can see we've set the centre of the normal distribution to 0.0. You can see the expression for the standard deviation related to the number of nodes in next

layer in Python form as pow(self.hnodes, -0.5) which is simply raising the number of nodes to the power of -0.5. That last parameter is the shape of the numpy array we want.

Querying the Network

It sounds logical to start working on the code that trains the neural network next, by fleshing out the currently empty train() function. We'll delay doing that, and instead, work on the simpler query() function. This will give us more time to gradually build confidence and get some practice at using both Python and these weight matrices inside the neural network object.

The query() function takes the input to a neural network and returns the network's output. That's simple enough, but to do that you'll remember that we need to pass the input signals from the input layer of nodes, through the hidden layer and out of the final output layer. You'll also remember that we use the link weights to moderate the signals as they feed into any given hidden or output node, and we also use the sigmoid activation function to squish the signal coming out of those nodes.

If we had lots of nodes, we would have a horrible task on our hands, writing out the Python code for each of those nodes, doing the weight moderating, summing signals, applying the activation function. And the more nodes, the more code. That would be a nightmare!

Luckily we don't have to do that because we worked out how to write all these instructions in a simple concise matrix form. The following shows how the matrix of weights for the link between the input and hidden layers can be combined with the matrix of inputs to give the signals into the hidden layer nodes:

$$\mathbf{X}_{\mathbf{hidden}} = \mathbf{W}_{\mathbf{input\_hidden}} \cdot \mathbf{I}$$

The great thing about this was not just that it was easier for us to write down, but that programming languages like Python can also understand matrices and do all the real work quite efficiently because they recognise the similarities between all the underlying calculations.

You'll be amazed at how simple the Python code actually is. The following applies the numpy library's dot product function for matrices to the link weights $\mathbf{W}_{\mathbf{input\_hidden}}$ and the inputs $\mathbf{I}$.

```
hidden_inputs = numpy.dot(self.wih, inputs)
```

That's it!

That simple piece of Python does all the work of combining all the inputs with all the right link weights to produce the matrix of combined moderated signals into each hidden layer node. We don't have to rewrite it either if next time we choose to use a different number of nodes for the input or hidden layers. It just works!

This power and elegance is why we put the effort into trying to understand the matrix multiplication approach earlier.

To get the signals emerging from the hidden node, we simply apply the sigmoid squashing function to each of these emerging signals:

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

That should be easy, especially if the sigmoid function is already defined in a handy Python library. It turns out it is! The scipy Python library has a set of special functions, and the sigmoid function is called expit(). Don't ask me why it has such a silly name. This scipy library is imported just like we imported the numpy library:

```
# scipy.special for the sigmoid function expit()
import scipy.special
```

Because we might want to experiment and tweak, or even completely change, the activation function, it makes sense to define it only once inside the neural network object when it is first initialised. After that we can refer to it several times, such as in the query() function. This arrangement means we only need to change this definition once, and not have to locate and change the code anywhere an activation function is used.

The following defines the activation function we want to use inside the neural network's initialisation section.

```
# activation function is the sigmoid function
self.activation_function = lambda x: scipy.special.expit(x)
```

What is this code? It doesn't look like anything we've seen before. What is that **lambda**? Well, this may look daunting, but it really isn't. All we've done here is created a function like any other, but we've used a shorter way of writing it out.

Instead of the usual def() definitions, we use the magic **lambda** to create a function there and then, quickly and easily. The function here takes x and returns scipy.special.expit(x) which is the sigmoid function. Functions created with lambda are nameless, or **anonymous** as seasoned coders like to call them, but here we've assigned it to the name self.activation_function(). All this means is that whenever someone needs to user the activation function, all they need to do is call self.activation_function().

So, going back to the task at hand, we want to apply the activation function to the combined and moderated signals into the hidden nodes. The code is as simple as the following:

```
# calculate the signals emerging from hidden layer
hidden_outputs = self.activation_function(hidden_inputs)
```

That is, the signals emerging from the hidden layer nodes are in the matrix called **hidden_outputs**.

That got us as far as the middle hidden layer, what about the final output layer? Well, there isn't anything really different between the hidden and final output layer nodes, so the process is the same. This means the code is also very similar.

Have a look at the following code which summarises how we calculate not just the hidden layer signals but also the output layer signals too.

```
# calculate signals into hidden layer
hidden_inputs = numpy.dot(self.wih, inputs)
# calculate the signals emerging from hidden layer
hidden_outputs = self.activation_function(hidden_inputs)

# calculate signals into final output layer
final_inputs = numpy.dot(self.who, hidden_outputs)
# calculate the signals emerging from final output layer
final_outputs = self.activation_function(final_inputs)
```

If we took away the comments, there are just four lines of code shown in bold that do all the calculations we needed, two for the hidden layer and two for the final output layer.

The Code Thus Far

Let's take a breath and pause to check what the code for the neural network class we're building up looks like. It should look something like the following.

```
# neural network class definition
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes,
learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to
node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
(self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
(self.onodes, self.hnodes))

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass


    # train the neural network
    def train():
        pass


    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
```

```
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs
```

That is just the class, aside from that we should be importing the numpy and scipy.special modules right at the top of the code in the first notebook cell:

```
import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
```

It is worth briefly noting the query() function only needs the **input_list**. It doesn't need any other input.

That's good progress, and now we look at the missing piece, the train() function. Remember there are two phases to training, the first is calculating the output just as query() does it, and the second part is backpropagating the errors to inform how the link weights are refined.

Before we go on to write the train() function for training the network with examples, let's just test the code we have at this point works. Let's create a small network and query it with some random input, just to see it working. Obviously there won't be any real meaning to this, we're only doing this to use these functions we just created.

The following shows the creation of a small network with 3 nodes in each of the input, hidden and output layers, and queries it with a randomly chosen input of (1.0, 0.5, -1.5).

```
In [3]: # number of input, hidden and output nodes
        input_nodes = 3
        hidden_nodes = 3
        output_nodes = 3

        # learning rate is 0.3
        learning_rate = 0.3

        # create instance of neural network
        n = neuralNetwork(input_nodes,hidden_nodes,output_nodes, learning_rate)

In [4]: n.query([1.0, 0.5, -1.5])

Out[4]: array([[ 0.45122712],
               [ 0.44630336],
               [ 0.49183299]])
```

You can see the creation of a neural network object does need a learning rate to be set, even though we're not using it yet. That's because our definition of the neural network class has an initialisation function __init__() that does require it to be specified. If it's not set, the Python code will fail and throw you an error to chew on!

You can also see that the input is a list, which in Python is written inside square brackets. The output is also a list, with some numbers. Even if this output has no real meaning, because we haven't trained the network, we can be happy the thing worked with no errors.

Training the Network

Let's now tackle the slightly more involved training task. There are two parts to this:

- The first part is working out the output for a given training example. That is no different to what we just did with the query() function.
- The second part is taking this calculated output, comparing it with the desired output, and using the difference to guide the updating of the network weights.

We've already done the first part so let's write that out:

```
# train the neural network
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
```

```
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    pass
```

This code is almost exactly the same as that in the query() function, because we're feeding forward the signal from the input layer to the final output layer in exactly the same way.

The only difference is that we have an additional parameter, **targets_list**, defined in the function name because you can't train the network without the training examples which include the desired or target answer.

```
def train(self, inputs_list, targets_list)
```

The code also turns the **targets_list** into a numpy array, just as the **inputs_list** is turned into a numpy array.

```
targets = numpy.array(targets_list, ndmin=2).T
```

Now we're getting closer to the heart of the neural network's working, improving the weights based on the error between the calculated and target output.

Let's do this in gentle manageable steps.

First we need to calculate the error, which is the difference between the desired target output provided by the training example, and the actual calculated output. That's the difference between the matrices (**targets - final_outputs**) done element by element. The Python code for this is really simple, showing again the elegant power of matrices.

```
# error is the (target - actual)
output_errors = targets - final_outputs
```

We can calculate the back-propagated errors for the hidden layer nodes. Remember how we split the errors according to the connected weights, and

recombine them for each hidden layer node. We worked out the matrix form of this calculation as

$$\mathbf{errors_{hidden} = weights^{T}_{hidden\_output} \cdot errors_{output}}$$

The code for this is again simple because of Python's ability to do matrix dot products using numpy.

```
# hidden layer error is the output_errors, split by weights, recombined at hidden nodes
hidden_errors = numpy.dot(self.who.T, output_errors)
```

So we have what we need to refine the weights at each layer. For the weights between the hidden and final layers, we use the **output_errors**. For the weights between the input and hidden layers, we use these **hidden_errors** we just calculated.

We previously worked out the expression for updating the weight for the link between a node **j** and a node **k** in the next layer in matrix form:

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

The alpha is the learning rate, and the sigmoid is the squashing activation function we saw before. Remember that the * multiplication is the normal element by element multiplication, and the · dot is the matrix dot product. That last bit, the matrix of outputs from the previous layer, is transposed. In effect this means the column of outputs becomes a row of outputs.

That should translate nicely into Python code. Let's do the code for the weights between the hidden and final layers first:

```
# update the weights for the links between the hidden and output layers
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))
```

That's a long line of code, but the colour coding should help you see how it relates back to that mathematical expression. The learning rate is self.**lr** and

simply multiplied with the rest of the expression. There is a matrix multiplication done by numpy.dot() and the two elements are coloured red and green to show the part related to the error and sigmoids from the next layer, and the transposed outputs from the previous layer.

That += simply means increase the preceding variable by the next amount. So **x** += 3 means increase **x** by 3. It's just a shorter way of writing **x** = **x** + 3. You can use other arithmetic too so **x** /= 3 means divide **x** by 3.

The code for the other weights between the input and hidden layers will be very similar. We just exploit the symmetry and rewrite the code replacing the names so that they refer to the previous layers. Here is the code for both sets of weights, coloured so you can see the similarities and differences:

```
# update the weights for the links between the hidden and output layers
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

# update the weights for the links between the input and hidden layers
self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))
```

That's it!

It's hard to believe that all that work we did earlier with huge volumes of calculations, the effort we put into working out the matrix approach, working our way through the gradient descent method of minimising the network error, … all that has amounted to the above short concise code! In some ways, that's the power of Python but actually it is a reflection of our hard work to simplify something which could easily have been made complex and horrible.

The Complete Neural Network Code

We've completed the neural network class. Here it is for reference, and you can always get it from the following link to github, an online place to sharing code:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

```
# neural network class definition
class neuralNetwork:
```

```
    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes,
learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to
node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
(self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
(self.onodes, self.hnodes))

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass

    # train the neural network
    def train(self, inputs_list, targets_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
```

```
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        # output layer error is the (target - actual)
        output_errors = targets - final_outputs
        # hidden layer error is the output_errors, split by weights, recombined
at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # update the weights for the links between the hidden and output
layers
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0
- final_outputs)), numpy.transpose(hidden_outputs))

        # update the weights for the links between the input and hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs *
(1.0 - hidden_outputs)), numpy.transpose(inputs))

        pass

    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs
```

There’s not a lot of code here, especially if you appreciate that this code can be used to create, train and query 3-layer neural networks for almost any task.

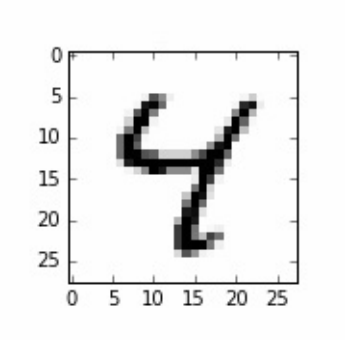
Next we’ll work on our specific task of learning to recognise numbers written by humans.

The MNIST Dataset of Handwritten Numbers

Recognising human handwriting is an ideal challenge for testing artificial intelligence, because the problem is sufficiently hard and fuzzy. It’s not clear and defined like multiplying lots of lots of numbers.

Getting computers to correctly classify what an image contains, sometimes called the **image recognition** problem, has withstood decades of attack. Only recently has good progress been made, and methods like neural networks have been a crucial part of these leaps forward.

To give you a sense of how hard the problem of image recognition is, we humans will sometimes disagree on what an image contains. We’ll easily disagree what a handwritten character actually is, particularly if the character was written in a rush or without care. Have a look at the following handwritten number. Is it a 4 or a 9?



There is a collection of images of handwritten numbers used by artificial intelligence researchers as a popular set to test their latest ideas and algorithms. The fact that the collection is well known and popular means that it is easy to check how well our latest crazy idea for image recognition works compared to others. That is, different ideas and algorithms are tested against the same data set.

That data set is called the MNIST database of handwritten digits, and is available from the respected neural network researcher Yann LeCun's website http://yann.lecun.com/exdb/mnist/. That page also lists how well old and new ideas have performed in learning and correctly classifying these handwritten characters. We'll come back to that list several times to see how well our own ideas perform against professionals!

The format of the MNIST database isn't the easiest to work with, so others have helpfully created data files of a simpler format, such as this one http://pjreddie.com/projects/mnist-in-csv/. These files are called CSV files, which means each value is plain text separated by commas (comma separated values). You can easily view them in any text editor, and most spreadsheet or data analysis software will work with CSV files. They are pretty much a universal standard. This website provides two CSV files:

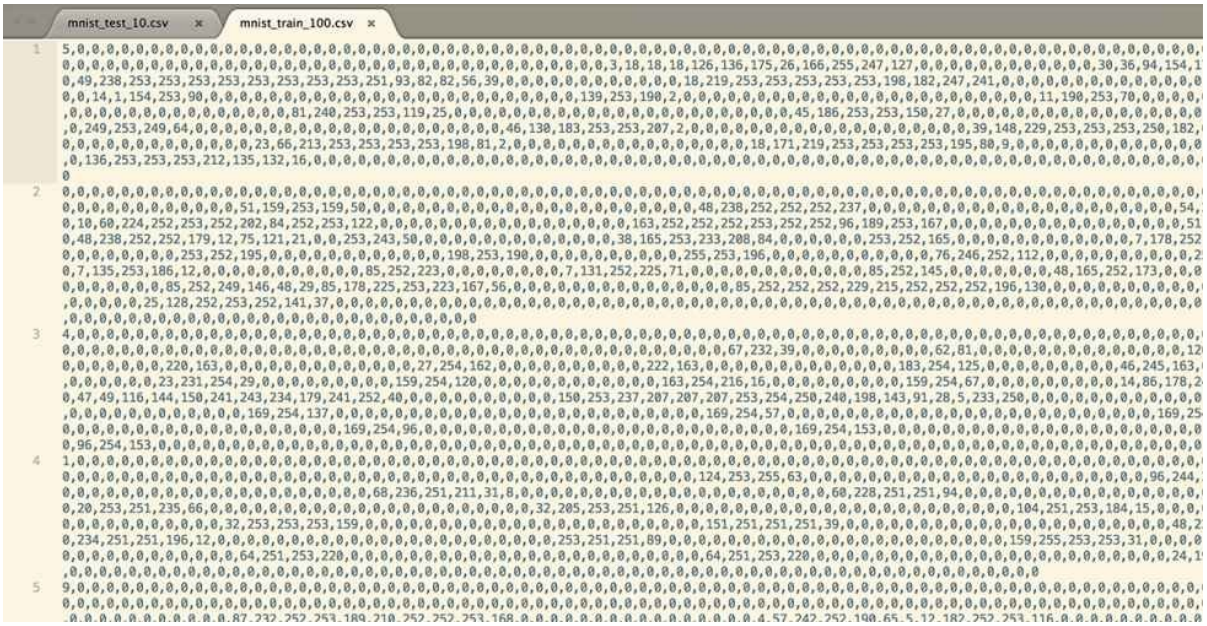
- A **training** set http://www.pjreddie.com/media/files/mnist_train.csv
- A **test** set http://www.pjreddie.com/media/files/mnist_test.csv

As the names suggest, the **training set** is the set of **60,000** labelled examples used to train the neural network. **Labelled** means the inputs come with the desired output, that is, what the answer should be.

The smaller **test set** of **10,000** is used to see how well our idea or algorithm works. This too contains the correct labels so we can check to see if our own neural network got the answer right or not.

The idea of having separate training and test data sets is to make sure we test against data we haven't seen before. Otherwise we could cheat and simply memorise the training data to get a perfect, albeit deceptive, score. This idea of separating training from test data is common across machine learning.

Let's take a peek at these files. The following shows a section of the MNIST test set loaded into a text editor.



Whoah! That looks like something went wrong! Like one of those movies from the 80s where a computer gets hacked.

Actually all is well. The text editor is showing long lines of text. Those lines consist of numbers, separated by commas. That is easy enough to see. The lines are quite long so they wrap around a few times. Helpfully this text editor shows the real line numbers in the margin, and we can see four whole lines of data, and part of the fifth one.

The content of these records, or lines of text, is easy to understand:

- The first value is the **label**, that is, the actual digit that the handwriting is supposed to represent, such as a "7" or a "9". This is the answer the neural network is trying to learn to get right.

- The subsequent values, all comma separated, are the pixel values of the handwritten digit. The size of the pixel array is 28 by 28, so there are 784 values after the label. Count them if you really want!

So that first record represents the number “5” as shown by the first value, and the rest of the text on that line are the pixel values for someone’s handwritten number 5. The second record represents a handwritten “0”, the third represents “4”, the fourth record is “1” and the fifth represents “9”. You can pick any line from the MNIST data files and the first number will tell you the label for the following image data.

But it is hard to see how that long list of 784 values makes up a picture of someone's handwritten number 5. We should plot those numbers as an image to confirm that they really are the colour values of handwritten number.

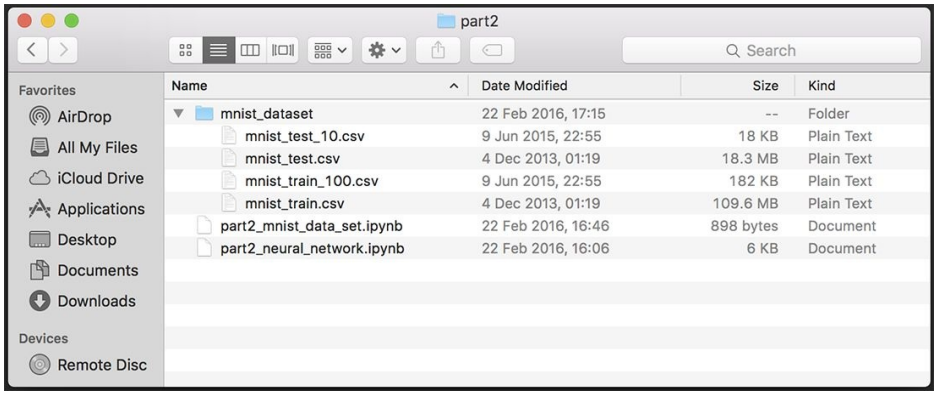
Before we dive in and do that we should download a smaller subset of the MNIST data set. The MNIST data data files are pretty big and working with a smaller subset is helpful because it means we can experiment, trial and develop our code without being slowed down by a large data set slowing our computers down. Once we've settled on an algorithm and code we're happy with, we can use the full data set.

The following are the links to a smaller subsets of the MNIST dataset, also in CSV format:

- 10 records from the MNIST test data set - https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyour
- 100 records from the MNIST training data set - https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyour

If your browser shows the data instead of downloading it automatically, you can manually save the file using the "File -> Save As …", or equivalent action in your browser.

Save the data files to a location that works well for you. I keep my data files in a folder called "mnist_dataset" next to my IPython notebooks as shown in the following screenshot. It gets messy if IPython notebooks and data files are scattered all over the place.



Before we can do anything with the data, like plotting it or training a neural network with it, we need to find a way to get at it from our Python code.

Opening a file and getting its content is really easy in Python. It's best to just show it and explain it. Have a look at the following code:

```
data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
data_list = data_file.readlines()
data_file.close()
```

There are only three lines of code here. Let's talk through each one.

The first line uses a function open() to open a file. You can see first parameter passed to the function is the name of the file. Actually, it is more than just the filename "mnist_train_100.csv", it is the whole path which includes the directory the file is in. The second parameter is optional, and tells Python how we want to treat the file. The 'r' tells Python that we want to open the file for reading only, and not for writing. That way we avoid any accidents changing or deleting the data. If we tried to write to that file and change it, Python would stop us and raise an error. What's that variable **data_file**? The open() function creates a file handle, a reference, to that file and we've assigned it to a variable named **data_file**. Now that we've opened the file, any further actions like reading from it, are done through that handle.

The next line is simple. We use the readlines() function associated with the file handle **data_file**, to read all of the lines in the file into the variable data_list. The variable contains a list, where each item of the list is a string representing a line in the file. That's much more useful because we can jump to specific lines just like we would jump to specific entries in a list. So **data_list**[0] is the first record, and **data_list**[9] is the tenth record, and so on.

By the way, you may hear people tell you not to use readlines() because it reads the entire file into memory. They will tell you to read one line at a time and do whatever work you need with each line, and then move onto the next line. They aren't wrong, it is more efficient to work on a line at a time, and not read the entire file into memory. However our files aren't that massive, and the code is easier if we use readlines(), and for us, simplicity and clarity is important as we learn Python.

The last line closes the file. It is good practice to close and clean up after using resources like files. If we don't, they remain open and can cause problems. What problems? Well, some programs might not want to write to a file that was opened elsewhere in case it led to an inconsistency. It would be like two people trying to write a letter on the same piece of paper! Sometimes your computer

may lock a file to prevent this kind of clash. If you don't clean up after you use files, you can have a build up of locked files. At the very least, closing a file, allows your computer to free up memory it used to hold parts of that file.

Create a new empty notebook and try this code, and see what happens when you print out elements of that list a. The following shows this working.

```
In [8]:  data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
         data_list = data_file.readlines()
         data_file.close()

In [9]:  len(data_list)

Out[9]:  100

In [10]: data_list[0]

Out[10]: '5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,18,18,18,126,136,175,26,166,255,247,127,0,
         0,0,0,0,0,0,0,0,0,0,0,30,36,94,154,170,253,253,253,253,253,225,172,253,242,195,64,0,0,0,0,0,0,0,0,0,0,0,49,238,253,25
         3,253,253,253,253,253,253,251,93,82,82,56,39,0,0,0,0,0,0,0,0,0,0,0,0,18,219,253,253,253,253,253,198,182,247,241,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,80,156,107,253,253,205,11,0,43,154,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,1,154,253,
         90,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,139,253,190,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11
         ,190,253,70,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,35,241,225,160,108,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,81,240,253,253,119,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,186,253,253,150,27,0,0,0,0,0,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,93,252,253,187,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,249,253,249,64,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,46,130,183,253,253,207,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,148,229,253,
         253,253,250,182,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,114,221,253,253,253,253,201,78,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
         ,0,23,66,213,253,253,253,253,198,81,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,171,219,253,253,253,253,195,80,9,0,0,0,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,55,172,226,253,253,253,253,244,133,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,136,253,253,253,212,13
         5,132,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\n'
```

You can see that the length of the list is 100. The Python len() function tells us how big a list is. You can also see the content of the first record **data_list**[0]. The first number is '5' which is the label, and the rest of the 784 numbers are the colour values for the pixels that make up the image. If you look closely you can tell these colour values seem to range between 0 and 255. You might want to look at other records and see if that is true there too. You'll find that the colour values do indeed fall within the range from 0 to 255.

We did see earlier how we might plot a rectangular array of numbers using the imshow() function. We want to do the same here but we need to convert that list of comma separated numbers into a suitable array. Here are the steps to do that:

- Split that long text string of comma separated values into individual values, using the commas as the place to do the splitting.
- Ignore the first value, which is the label, and take the remaining list of 28 * 28 = 784 values and turn them into an array which has a shape of 28 rows by 28 columns.
- Plot that array!

Again, it is easiest to show the fairly simple Python code that does this, and talk through the code to explain in more detail what is happening.

First we mustn't forget to import the Python extension libraries which will help us with arrays and plotting:

```
import numpy
import matplotlib.pyplot
%matplotlib inline
```

Look at the following 3 lines of code. The variables have been coloured to make it easier to understand which data is being used where.

```
all_values = data_list[0].split(',')
image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys',
interpolation='None')
```

The first line takes the first records **data_list**[0] that we just printed out, and splits that long string by its commas. You can see the split() function doing this, with a parameter telling it which symbol to split by. In this case that symbol is the comma. The results will be placed into **all_values**. You can print this out to check that is indeed a long Python list of values.

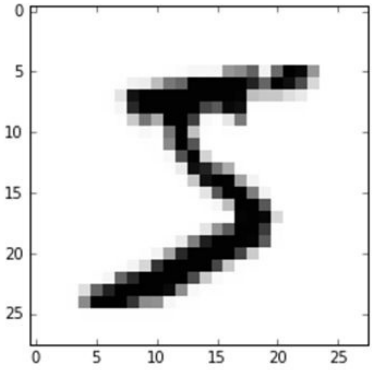
The next line looks more complicated because there are several things happening on the same line. Let's work out from the core. The core has that **all_values** list but this time the square brackets [1:] are used to take all except the first element of this list. This is how we ignore the first label value and only take the rest of the 784 values. The numpy.asfarray() is a numpy function to convert the text strings into real numbers and to create an array of those numbers. Hang on - what do we mean by converting text string into numbers? Well the file was read as text, and each line or record is still text. Splitting each line by the commas still results in bits of text. That text could be the word "apple", "orange123" or "567". The text string "567" is not the same as a number 567. That's why we need to convert text strings to numbers, even if the text looks like numbers. The last bit .reshape((28,28)) makes sure the list of number is wrapped around every 28 elements to make a square matrix 28 by 28. The resulting 28 by 28 array is called image_array. Phew! That was a fair bit happening in one line.

The third line simply plots the image_array using the imshow() function just like we saw earlier. This time we did select a greyscale colour palette with

cmap=’Greys’ to better show the handwritten characters.

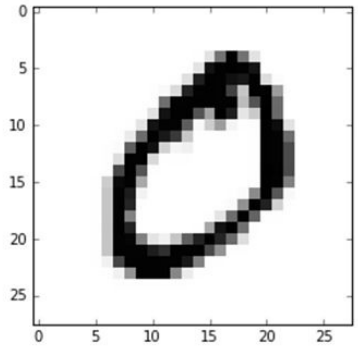
The following shows the results of this code:

```
In [32]: all_values = data_list[0].split(',')
         image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
         matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[32]: <matplotlib.image.AxesImage at 0x108818cc0>
```



You can see the plotted image is a 5, which is what the label said it should be. If we instead choose the next record data_list[1] which has a label of 0, we get the following image.

```
In [37]: all_values = data_list[1].split(',')
         image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
         matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[37]: <matplotlib.image.AxesImage at 0x108bc3160>
```



You can tell easily the handwritten number is indeed zero.

Preparing the MNIST Training Data

We’ve worked out how to get data out of the MNIST data files and disentangle it so we can make sense of it, and visualise it too. We want to train our neural network with this data, but we need to think just a little about preparing this data before we throw it at our neural network.

We saw earlier that neural networks work better if the input data, and also the output values, are of the right shape so that they stay within the comfort zone of the network node activation functions.

The first thing we need to do is to rescale the input colour values from the larger range 0 to 255 to the much smaller range 0.01 - 1.0. We've deliberately chosen 0.01 as the lower end of the range to avoid the problems we saw earlier with zero valued inputs because they can artificially kill weight updates. We don't have to choose 0.99 for the upper end of the input because we don't need to avoid 1.0 for the inputs. It's only for the outputs that we should avoid the impossible to reach 1.0.

Dividing the raw inputs which are in the range 0-255 by 255 will bring them into the range 0-1. We then need to multiply by 0.99 to bring them into the range 0.0 - 0.99. We then add 0.01 to shift them up to the desired range 0.01 to 1.00. The following Python code shows this in action:

```
scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
print(scaled_input)
```

The output confirms that the values are now in the range 0.01 to 0.99.

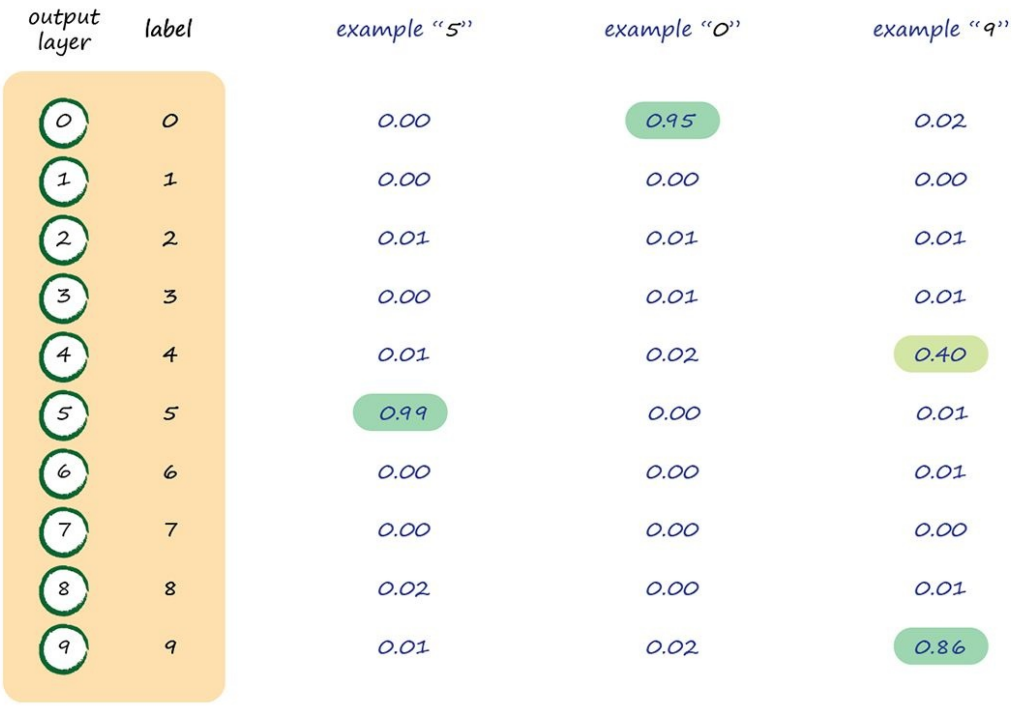
```
In [19]: # scale input to range 0.01 to 1.00
         scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
         print(scaled_input)

[ 0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.208       0.62729412  0.99223529  0.62729412  0.20411765
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.19635294  0.934       0.98835294  0.98835294  0.98835294
  0.93011765  0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.21964706  0.89129412  0.99223529  0.98835294  0.93788235
  0.91458824  0.98835294  0.23129412  0.03329412  0.01        0.01        0.01
```

So we've prepared the MNIST data by rescaling and shifting it, ready to throw at our neural network for both training and querying.

We now need to think about the neural network's outputs. We saw earlier that the outputs should match the range of values that the activation function can push out. The logistic function we're using can't push out numbers like -2.0 or 255. The range is between 0.0 and 1.0, and in fact you can't reach 0.0 or 1.0 as the logistic function only approaches these extremes without actually getting there. So it looks like we'll have to scale our target values when training.

But actually, we have a deeper question to ask ourselves. What should the output even be? Should it be an image of the answer? That would mean we have a 28x28 = 784 output nodes.

If we take a step back and think about what we're asking the neural network, we realise we're asking it to classify the image and assign the correct label. That label is one of 10 numbers, from 0 to 9. That means should be able to have an output layer of 10 nodes, one for each of the possible answers, or labels. If the answer was "0" the first output layer node would fire and the rest should be silent. If the answer was "9" the last output layer node would fire and the rest would be silent. The following illustrates this scheme, with some example outputs too.



The first example is where the neural network thinks it has seen the number "5". You can see that the largest signal emerging from the output layer is from the node with label 5. Remember this is the sixth node because we're starting from a

label for zero. That's easy enough. The rest of the output nodes produce a small signal very close to zero. Rounding errors might result in an output of zero, but in fact you'll remember the activation function doesn't produce an actual zero.

The next example shows what might happen if the neural network thinks it has seen a handwritten "zero". Again the largest output by far is from the first output node corresponding to the label "0'.

The last example is more interesting. Here the neural network has produced the largest output signal from the last node, corresponding to the label "9". However it has a moderately big output from the node for "4". Normally we would go with the biggest signal, but you can see how the network partly believed the answer could have been "4". Perhaps the handwriting made it hard to be sure? This sort of uncertainty does happen with neural networks, and rather than see it as a bad thing, we should see it as a useful insight into how another answer was also a contender.

That's great! Now we need to turn these ideas into target arrays for the neural network training. You can see that if the label for a training example is "5", we need to create a target array for the output node where all the elements are small except the one corresponding to the label "5". That could look like the following [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].

In fact we need to rescale those numbers because we've already seen how trying to get the neural network to create outputs of 0 and 1, which are impossible for the activation function, will drive large weights and a saturated network. So we'll use the values 0.01 and 0.99 instead, so the target for the label "5" should be [0.01, 0.01, 0.01, 0.01, 0.01, 0.99, 0.01, 0.01, 0.01, 0.01].

Have a look at the following Python code which constructs the target matrix:

```
#output nodes is 10 (example)
onodes = 10
targets = numpy.zeros(onodes) + 0.01
targets[int(all_values[0])] = 0.99
```

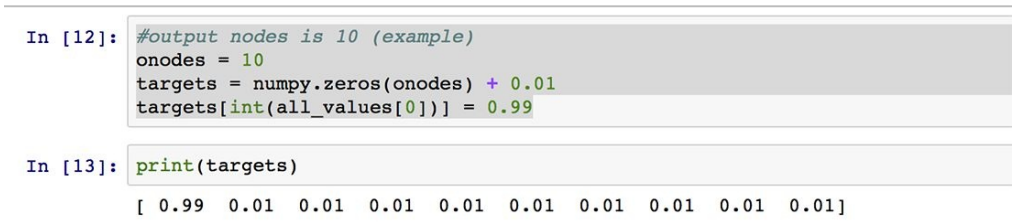
The first line, other than the comment, simply sets the number of output nodes to 10, which is right for our example with ten labels.

The second line simple uses a convenient numpy function called numpy.zeros() to create an array filled with zeros. The parameter it takes is the size and shape

of the array we want. Here we just want a simple one of length **onodes**, which the number of nodes on the final output layer. We add 0.01 to fix the problem with zeros we just talked about.

The next line takes the first element of the MNIST dataset record, which is the training target label, and converts that string into an integer. Remember that record is read from the source files as a text string, not a number. Once that conversion is done, that target label is used to set the right element of the targets list to 0.99. It looks neat because a label of "0" will be converted to integer 0, which is the correct index into the **targets**[] array for that label. Similarly, a label of "9" will be converted to integer 9, and **targets**[9] is indeed the last element of that array.

The following shows an example of this working:



Excellent, we have now worked out how to prepare the inputs for training and querying, and the outputs for training too.

Let's update our Python code to include this work. The following shows the code developed thus far. The code will always be available on github at the following link, but will evolve as we add more to it:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

You can always see the previous versions as they've developed at the following history view:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

python notebook for Make Your Own Neural Network
code for a 3-layer neural network, and code for learning the MNIST dataset
(c) Tariq Rashid, 2016
license is GPLv2

```
import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
# library for plotting arrays
import matplotlib.pyplot
# ensure the plots are inside this notebook, not an external window
%matplotlib inline

# neural network class definition
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to
node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
(self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
(self.onodes, self.hnodes))

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass


    # train the neural network
```

```
    def train(self, inputs_list, targets_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        # output layer error is the (target - actual)
        output_errors = targets - final_outputs
        # hidden layer error is the output_errors, split by weights, recombined
at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # update the weights for the links between the hidden and output
layers
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0
- final_outputs)), numpy.transpose(hidden_outputs))

        # update the weights for the links between the input and hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs *
(1.0 - hidden_outputs)), numpy.transpose(inputs))

        pass


    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
```

```
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs

# number of input, hidden and output nodes
input_nodes = 784
hidden_nodes = 100
output_nodes = 10

# learning rate is 0.3
learning_rate = 0.3

# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
learning_rate)

# load the mnist training data CSV file into a list
training_data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# train the neural network

# go through all records in the training data set
for record in training_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # create the target output values (all 0.01, except the desired label which
is 0.99)
    targets = numpy.zeros(output_nodes) + 0.01
    # all_values[0] is the target label for this record
    targets[int(all_values[0])] = 0.99
```

```
    n.train(inputs, targets)
    pass
```

You can see we've imported the plotting library at the top, added some code to set the size of the input, hidden and output layers, read the smaller MNIST training data set, and then trained the neural network with those records.

Why have we chosen 784 input nodes? Remember, that's 28 x 28, the pixels which make up the handwritten number image.

The choice of 100 hidden nodes is not so scientific. We didn't choose a number larger than 784 because the idea is that neural networks should find features or patterns in the input which can be expressed in a shorter form than the input itself. So by choosing a value smaller than the number of inputs, we force the network to try to summarise the key features. However if we choose too few hidden layer nodes, then we restrict the ability of the network to find sufficient features or patterns. We'd be taking away its ability to express its own understanding of the MNIST data. Given the output layer needs 10 labels, hence 10 output nodes, the choice of an intermediate 100 for the hidden layer seems to make sense.

It is worth making an important point here. There isn't a perfect method for choosing how many hidden nodes there should be for a problem. Indeed there isn't a perfect method for choosing the number of hidden layers either. The best approaches, for now, are to experiment until you find a good configuration for the problem you're trying to solve.

Testing the Network

Now that we've trained the network, at least on a small subset of 100 records, we want to test how well that worked. We do this against the second data set, the training dataset.

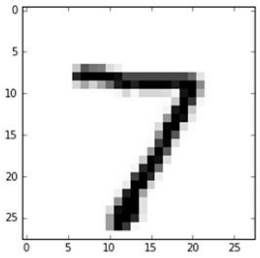
We first need to get at the test records, and the Python code is very similar to that used to get the training data.

```
# load the mnist test data CSV file into a list
test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
```

We unpack this data in the same way as before, because it has the same structure.

Before we create a loop to go through all the test records, let's just see what happens if we manually run one test. The following shows the first record from the test data set being used to query the now trained neural network.

```
In [27]: # load the mnist test data CSV file into a list
         test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r')
         test_data_list = test_data_file.readlines()
         test_data_file.close()

In [39]: # get the first test record
         all_values = test_data_list[0].split(',')
         # print the label
         print(all_values[0])

         7

In [40]: image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
         matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[40]: <matplotlib.image.AxesImage at 0x1090d4fd0>
```




```
In [41]: n.query((numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01)

Out[41]: array([[ 0.07652418],
                [ 0.01745079],
                [ 0.0054554 ],
                [ 0.07442751],
                [ 0.07348178],
                [ 0.01906993],
                [ 0.00938124],
                [ 0.7704694 ],
                [ 0.08000447],
                [ 0.05209131]])
```

You can see that the label for the first record from the test data set is "7". That's what we hope the neural network will answer when we query it.

Plotting the pixel values as an image confirms the handwritten number is indeed a "7".

Querying the trained network produces a list of numbers, the outputs from each of the output nodes. You can quickly see that one output value is much larger than the others, and is the one corresponding to the label "7". That's the eighth element, because the first one corresponds to the label "0".

It worked!

This is real moment to savour. All our hard work throughout this guide was worth it!

We trained our neural network and we just got it to tell is what it thinks is the number represented by that picture. Remember that it hasn't seen that picture before, it wasn't part of the training data set. So the neural network was able to correctly classify a handwritten character that it had not seen before. That is massive!

With just a few lines of simple Python we have created a neural network that learns to do something that many people would consider artificial intelligence - it learns to recognise images of human handwriting.

This is even more impressive given that we only trained on a tiny subset of the full training data set. Remember that training data set has 60,000 records, and we only trained on 100. I personally didn't think it would work!

Let's crack on and write the code to see how the neural network performs against the rest of the data set, and keep a score so we can later see if our own ideas for improving the learning worked, and also to compare with how well others have done this.

It's easiest to look at the following code and talk through it:

```
# test the neural network

# scorecard for how well the network performs, initially empty
scorecard = []

# go through all the records in the test data set
for record in test_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # correct answer is first value
    correct_label = int(all_values[0])
    print(correct_label, "correct label")
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    print(label, "network's answer")
    # append correct or incorrect to list
```

```
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to scorecard
        scorecard.append(0)
        pass

    pass
```

Before we jump into the loop which works through all the test data set records, we create an empty list, called **scorecard**, which will be the scorecard that we update after each record.

You can see that inside the loop, we do what we did before, we split the text record by the commas to separate out the values. We keep a note of the first value as the correct answer. We grab the remaining values and rescale them so they're suitable for querying the neural network. We keep the response from the neural network in a variable called **outputs**.

Next is the interesting bit. We know the output node with the largest value is the one the network thinks is the answer. The index of that node, that is, its position, corresponds to the label. That's a long way of saying, the first element corresponds to the label "0", and the fifth element corresponds to the label "4", and so on. Luckily there is a convenient numpy function that finds the largest value in an array and tells us its position, numpy.argmax(). You can read about it online here. If it returns 0 we know the network thinks the answer is zero, and and so on.

That last bit of code compares the label with the known correct label. If they are the same, a "1" is appended to the scorecard, otherwise a "0" is appended.

I've included some useful print() commands in the code so that we can see for ourselves the correct and predicted labels. The following shows the results of this code, and also of printing out the scorecard.

```
7 correct label
7 network's answer
2 correct label
0 network's answer
1 correct label
1 network's answer
0 correct label
0 network's answer
4 correct label
4 network's answer
1 correct label
1 network's answer
4 correct label
4 network's answer
9 correct label
4 network's answer
5 correct label
4 network's answer
9 correct label
7 network's answer
```

```
In [49]: print(scorecard)

[1, 0, 1, 1, 1, 1, 1, 0, 0, 0]
```

Not so great this time! We can see that there are quite a few mismatches. The final scorecard shows that out of ten test records, the network got 6 right. That's a score of 60%. That's not actually too bad given the small training set we used.

Let's finish off with some code to print out that test score as a fraction.

```
# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() / scorecard_array.size)
```

This is a simple calculation to workout the fraction of correct answers. It's the sum of "1" entries on the scorecard divided by the total number of entries, which is the size of the scorecard. Let's see what this produces.

```
In [49]: print(scorecard)

[1, 0, 1, 1, 1, 1, 1, 0, 0, 0]

In [59]: # calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() / scorecard_array.size)

performance =  0.6
```

That produces the fraction 0.6 or 60% accuracy as we expected.

Training and Testing with the Full Datasets

Let's add all this new code we've just developed to testing the network's performance to our main program.

While we're at it, let's change the file names so that we're now pointing to the full training data set of 60,000 records, and the test data set of 10,000 records. We previously saved those files as **mnist_dataset/mnist_train.csv** and **mnist_dataset/mnist_test.csv**. We're getting serious now!

Remember, you can get the Python notebook online at github:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

The history of that code is also available on github so you can see the code as it developed:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

The result of training our simple 3-layer neural network against the full 60,000 training examples, and then testing it against the 10,000 records, gives us an overall performance score of **0.9473**. That is very very good. Almost 95% accurate!

```
In [72]: # calculate the performance score, the fraction of correct answers
         scorecard_array = numpy.asarray(scorecard)
         print ("performance = ", scorecard_array.sum() / scorecard_array.size)

         performance =  0.9473
```

It is worth comparing this score of just under 95% accuracy against industry benchmarks recorded at http://yann.lecun.com/exdb/mnist/. We can see that we're better than some of the historic benchmarks, we are about the same performance as the simplest neural network approach listen there, which has a performance of 95.3%.

That's not bad at all. We should be very pleased that our very first go at a simple neural network achieves the kind of performance that a professional neural network researcher achieved.

By the way, it shouldn't surprise you that crunching through 60,000 training examples, each requiring a set of feedforward calculations from 784 input nodes, through 100 hidden nodes, and also doing an error feedback and weight update, all takes a while even for a fast modern home computer. My new laptop took about 2 minutes to get through the training loop. Yours may be quicker or slower.

Some Improvements: Tweaking the Learning Rate

A 95% performance score on the MNIST dataset with our first neural neural network, using only simple ideas and simple Python is not a bad at all, and if you wanted to stop here you would be entirely justified.

But let's see if we can make some easy improvements.

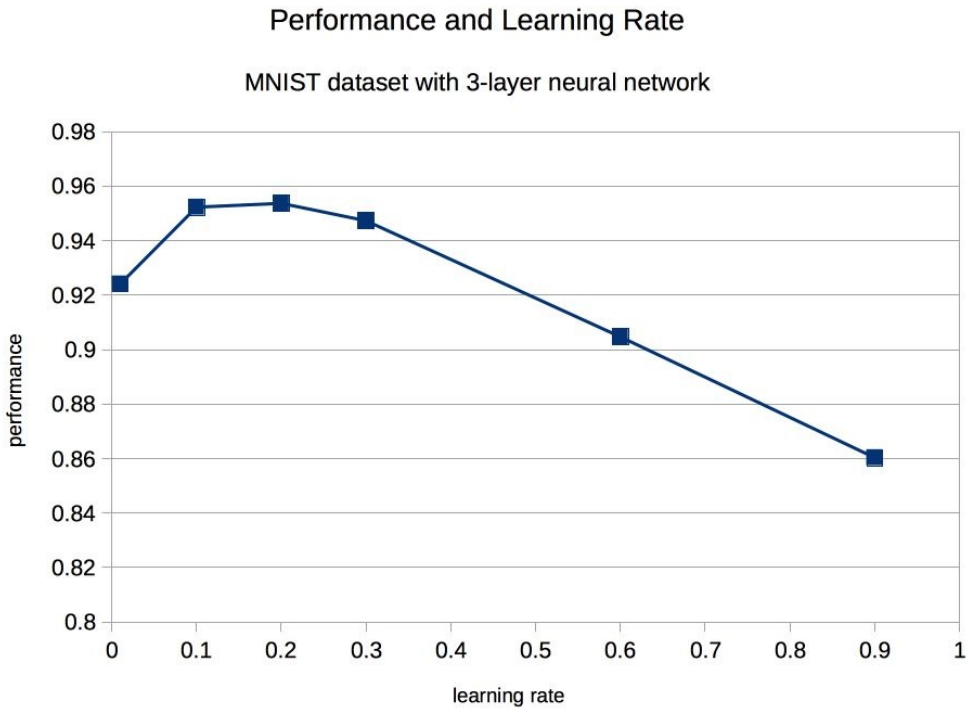
The first improvement we can try is to adjust the learning rate. We set it at 0.3 previously without really experimenting with different values.

Let's try doubling it to **0.6**, to see if a boost will actually be helpful or harmful to the overall network learning. If we run the code we get a performance score of **0.9047**. That's worse than before. So it looks like the larger learning rate leads to some bouncing around and overshooting during the gradient descent.

Let's try again with a learning rate of **0.1**. This time the performance is an improvement at **0.9523**. It's similar in performance to one listed on that website which has 1000 hidden nodes. We're doing well with much less!

What happens if we keep going and set a learning rate of an even smaller **0.01**? The performance isn't so good at **0.9241**. So it seems having too small a learning rate is damaging. This makes sense because we're limiting the speed at which gradient descent happens, we're making the steps too small.

The following plots a graph of these results. It's not a very scientific approach because we should really do these experiments many times to reduce the effect of randomness and bad journeys down the gradient descent, but it is still useful to see the general idea that there is a sweet spot for learning rate.



The plot suggested that between a learning rate of 0.1 and 0.3 there might be better performance, so let's try a learning rate of **0.2.** The performance is **0.9537**. That is indeed a tiny bit better than either 0.1 and 0.3 This idea of plotting graphs to get a better feel for what is going on is something you should consider in other scenarios too - pictures help us understand much better than a list of numbers!

So we'll stick with a learning rate of 0.2, which seems to be a sweet spot for the MNIST data set and our neural network.

By the way, if you run this code yourself, your own scores will be slightly different because the whole process is a little random. Your initial random weights won't be the same as my initial random weights, and so your own code will take a different route down the gradient descent than mine.

Some Improvements: Doing Multiple Runs

The next improvement we can do is to repeat the training several times against the data set. Some people call each run through an **epoch**. So a training session with 10 epochs means running through the entire training data set 10 times. Why would we do that? Especially if the time our computers take goes up to 10 or 20 or even 30 minutes? The reason it is worth doing is that we're helping those weights do that gradient descent by providing more chances to creep down those slopes.

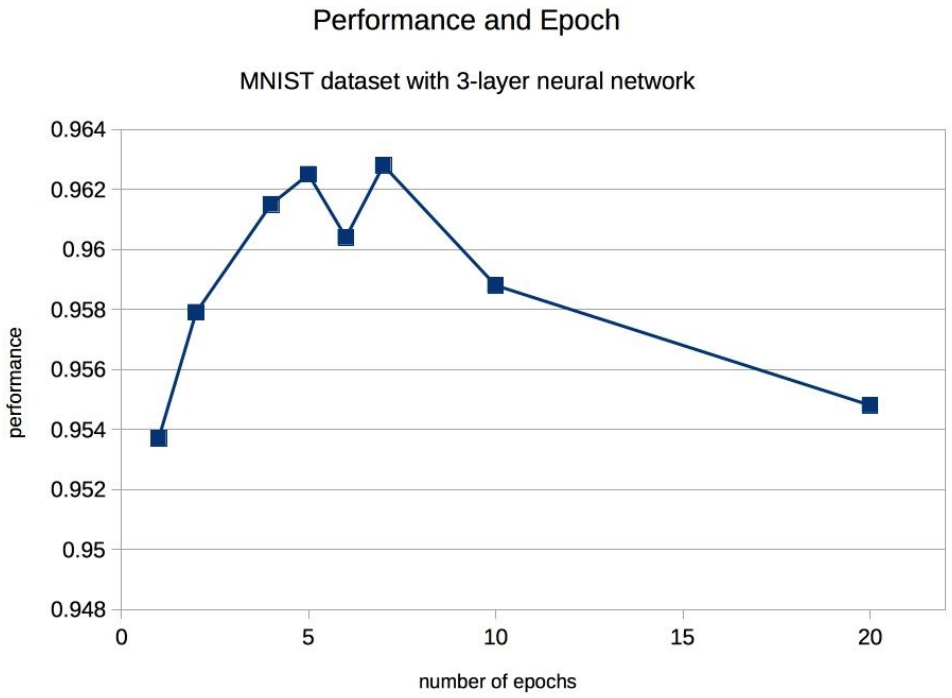
Let's try it with 2 epochs. The code changes slightly because we now add an extra loop around the training code. The following shows this outer loop colour coded to help see what's happening.

```
# train the neural network

# epochs is the number of times the training data set is used for training
epochs = 10

for e in range(epochs):
    # go through all records in the training data set
    for record in training_data_list:
        # split the record by the ',' commas
        all_values = record.split(',')
        # scale and shift the inputs
        inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        # create the target output values (all 0.01, except the desired label
which is 0.99)
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] is the target label for this record
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
        pass
    pass
```

The resulting performance with 2 epochs is **0.9579**, a small improvement over just 1 epoch.

Just like we did with tweaking the learning rate, let's experiment with a few different epochs and plot a graph to visualise the effect this has. Intuition suggests the more training you do the better the performance. Some of you will realise that too much training is actually bad because the network overfits to the training data, and then performs badly against new data that it hasn't seen before. This **overfitting** is something to beware of across many different kinds of machine learning, not just neural networks.

Here's what happens:

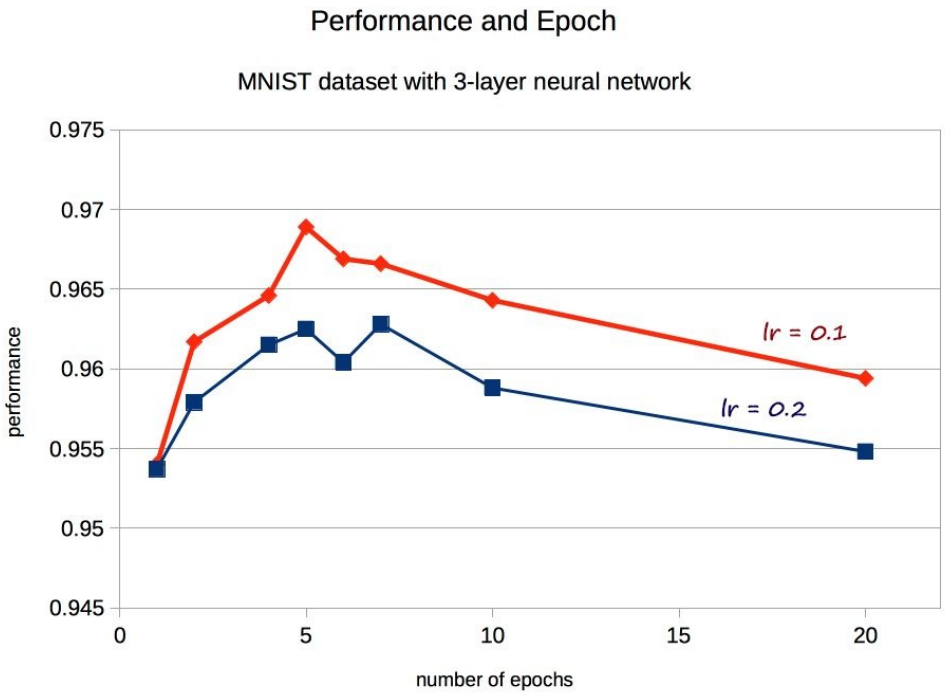


You can see the results aren't quite so predictable. You can see there is a sweet spot around 5 or 7 epochs. After that performance degrades, and this may be the effect of overfitting. The dip at 6 epochs is probably a bad run with the network getting stuck in a bad minimum during gradient descent. Actually, I would have expected much more variation in the results because we've not done many experiments for each data point to reduce the effect of expected variations from what is essentially a random process. That's why I've left that odd point for 6 epochs in, to remind us that neural network learning is a random process at heart and can sometimes not work so well, and sometimes work really badly.

Another idea is that the learning rate is too high for larger numbers of epochs. Lets try this experiment again and tune down the learning rate from 0.2 down to 0.1 and see what happens.

The peak performance is now up to **0.9628**, or 96.28%, with 7 epochs.

The following graph shows the new performance with learning rate at 0.1 overlaid onto the previous one.



You can see that calming down the learning rate did indeed produce better performance with more epochs. That peak of **0.9689** represents an approximate error rate of 3%, which is comparable to the networks benchmarks on Yann LeCun's website.

Intuitively it makes sense that if you plan to explore the gradient descent for much longer (more epochs), you can afford to take shorter steps (learning rate), and overall you'll find a better path down. It does seem that 5 epochs is probably the sweet spot for this neural network against this MNIST learning task. Again keep in mind that we did this in a fairly unscientific way. To do it properly you would have to do this experiment many times for each combination of learning rates and epochs to minimise the effect of randomness that is inherent in gradient descent.

Change Network Shape

One thing we haven't yet tried, and perhaps should have earlier, is to change the shape of the neural network. Let's try changing the number of middle hidden layer nodes. We've had them set to 100 for far too long!

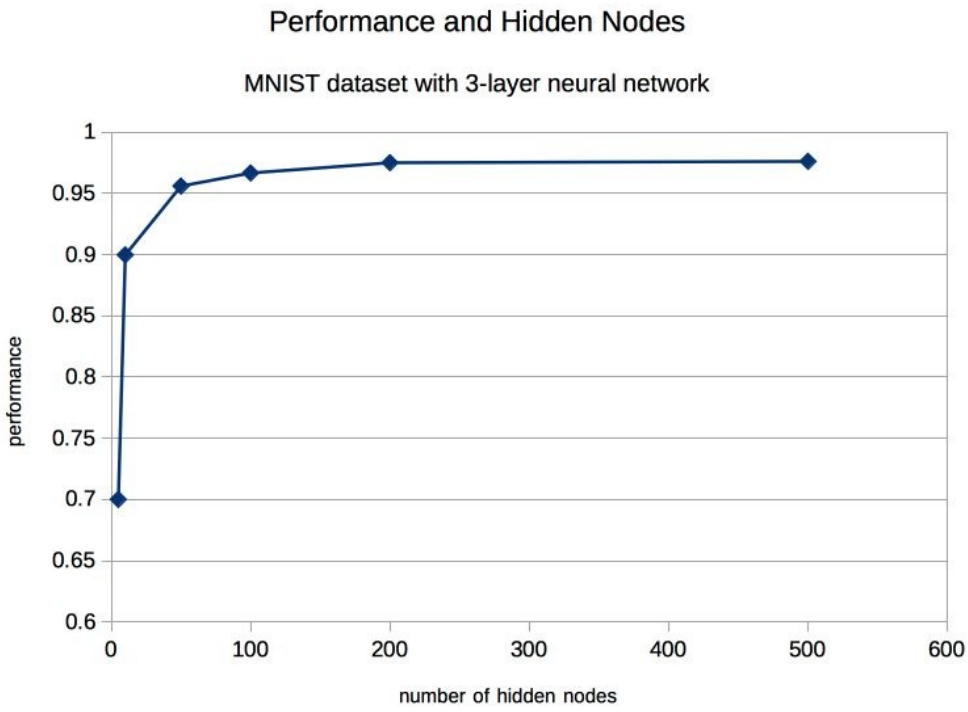
Before we jump in and run experiments with different numbers of hidden nodes, let's think about what might happen if we do. The hidden layer is the layer which is where the learning happens. Remember the input nodes simply bring in

the input signals, and the output nodes simply push out the network's answer. It's the hidden layer (or layers) which have to learn to turn the input into the answer. It's where the learning happens. Actually, it's the link weights before and after the hidden nodes that do the learning, but you you know what I mean.

If we had too few hidden nodes, say 3, you can image there is no way there is enough space to learn whatever a network learns, to somehow turn all the inputs into the correct outputs. It would be like asking a car with 5 seats to carry 10 people. You just can't fit that much stuff inside. Computer scientists call this kind of limit a **learning capacity**. You can't learn more than the learning capacity, but you can change the vehicle, or the network shape, to increase the capacity.

What if we had 10000 hidden nodes? Well we won't be short of learning capacity, but we might find it harder to train the network because now there are too many options for where the learning should go. Maybe it would take 10000s of epochs to train such a network.

Let's run some experiments and see what happens.



You can see that for low numbers of hidden nodes the results are not as good for higher numbers. We expected that. But the performance from just 5 hidden

nodes was **0.7001**. That is pretty amazing given that from such few learning locations the network is still about 70% right. Remember we've been running with 100 hidden nodes thus far. Just 10 hidden nodes gets us **0.8998** accuracy, which again is pretty impressive. That's 1/10th of the nodes we've been used to, and the network's performance jumps to 90%.

This point is worth appreciating. The neural network is able to give really good results with so few hidden nodes, or learning locations. That's a testament to their power.

As we increase the number of hidden nodes, the results do improve but not as drastically. The time taken to train the network also increases significantly, because each extra hidden node means new network links to every node in the preceding and next layers, which all require lots more calculations! So we have to choose a number of hidden nodes with a tolerable run time. For my computer that's 200 nodes. Your computer may be faster or slower.

We've also set a new record for accuracy, with **0.9751** with 200 nodes. And a long run with 500 nodes gave us **0.9762**. That's really good compared to the benchmarks listed on LeCun's website.

Looking back at the graphs you can see that the previous stubborn limit of about 95% accuracy was broken by changing the shape of the network.

Good Work!

Looking back over this work, we've created a neural network using only the simple concepts we covered earlier, and using simple Python.

And that neural network has performed so well, without any extra fancy mathematical magic, its performance is very respectable compared to networks that academics and researchers make.

There's more fun in part three of guide, but even if you don't explore those ideas, don't hesitate to experiment further with the neural network you've already made - try a different number of hidden nodes, or a different scaling, or even a different activation function, just to see what happens.

Final Code

The following shows the final code in case you can't access the code on github, or just prefer a copy here for easy reference.

```
# python notebook for Make Your Own Neural Network
# code for a 3-layer neural network, and code for learning the MNIST dataset
# (c) Tariq Rashid, 2016
# license is GPLv2


import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
# library for plotting arrays
import matplotlib.pyplot
# ensure the plots are inside this notebook, not an external window
%matplotlib inline

# neural network class definition
class neuralNetwork:



    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5), (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5), (self.onodes, self.hnodes))

        # learning rate
        self.lr = learningrate
```

```
        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass

    # train the neural network
    def train(self, inputs_list, targets_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        # output layer error is the (target - actual)
        output_errors = targets - final_outputs
        # hidden layer error is the output_errors, split by weights, recombined
at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # update the weights for the links between the hidden and output
layers
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0
- final_outputs)), numpy.transpose(hidden_outputs))

        # update the weights for the links between the input and hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs *
(1.0 - hidden_outputs)), numpy.transpose(inputs))

        pass
```

```
    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs

# number of input, hidden and output nodes
input_nodes = 784
hidden_nodes = 200
output_nodes = 10

# learning rate
learning_rate = 0.1

# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
learning_rate)

# load the mnist training data CSV file into a list
training_data_file = open("mnist_dataset/mnist_train.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# train the neural network

# epochs is the number of times the training data set is used for training
epochs = 5
```

```
for e in range(epochs):
    # go through all records in the training data set
    for record in training_data_list:
        # split the record by the ',' commas
        all_values = record.split(',')
        # scale and shift the inputs
        inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        # create the target output values (all 0.01, except the desired label
which is 0.99)
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] is the target label for this record
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
        pass
    pass

# load the mnist test data CSV file into a list
test_data_file = open("mnist_dataset/mnist_test.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()

# test the neural network

# scorecard for how well the network performs, initially empty
scorecard = []

# go through all the records in the test data set
for record in test_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # correct answer is first value
    correct_label = int(all_values[0])
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    # append correct or incorrect to list
```

```
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to scorecard
        scorecard.append(0)
        pass

    pass

# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() / scorecard_array.size)
```

Part 3 - Even More Fun

“If you don’t play, you don’t learn.”

In this part of the guide we'll explore further ideas just because they're fun. They aren't necessary to understanding the basic of neural networks so don't feel you have to understand everything here.

Because this is a fun extra section, the pace will be slightly quicker, but we will still try to explain the ideas in plain English.

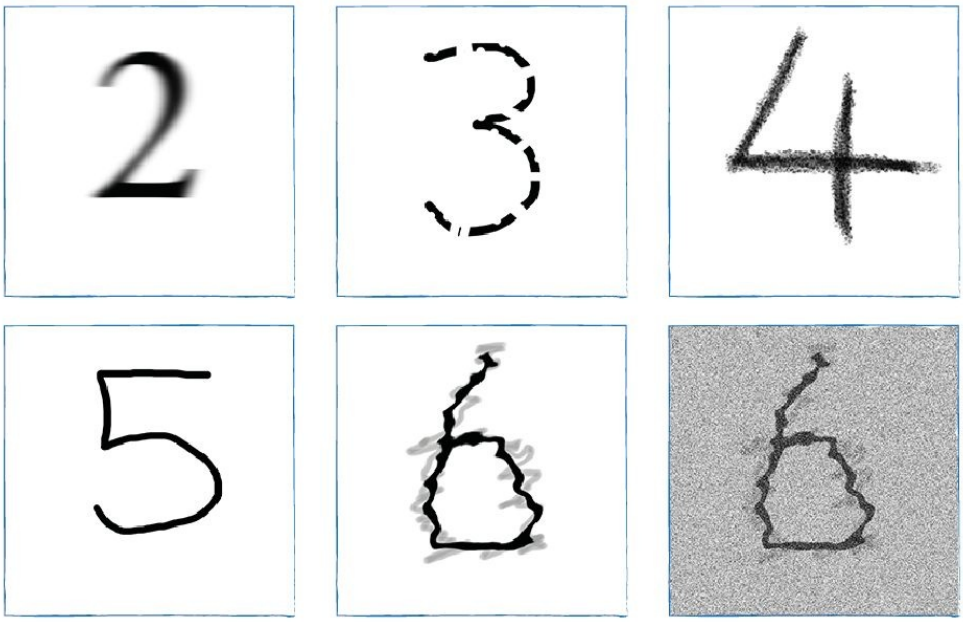
Your Own Handwriting

Throughout this guide we've been using images of handwritten numbers from the MNIST dataset. Why not use your own handwriting?

In this experiment, we'll create our test dataset using our own handwriting. We'll also try using different styles of writing, and noisy or shaky images to see how well our neural network copes.

You can create images using any image editing or painting software you like. You don't have to use the expensive Photoshop, the GIMP is a free open source alternative available for Windows, Mac and Linux. You can even use a pen on paper and photograph your writing with a smartphone or camera, or even use a proper scanner. The only requirement is that the image is square (the width is the same as the length) and you save it as PNG format. You'll often find the saving format option under File > Save As, or File > Export in your favourite image editor.

Here are some images I made.



The number 5 is simply my own handwriting. The 4 is done using a chalk rather than a marker. The number 3 is my own handwriting but deliberately with bits chopped out. The 2 is a very traditional newspaper or book typeface but blurred a bit. The 6 is a deliberately wobbly shaky image, almost like a reflection in water. The last image is the same as the previous one but with noise added to see if we can make the neural network's job even harder!

This is fun but there is a serious point here. Scientists have been amazed at the human brain's ability to continue to function amazingly well after suffering damage. The suggestion is that neural networks distribute what they've learned across several link weights, which means if they suffer some damage, they can perform fairly well. This also means that they can perform fairly well if the input image is damaged or incomplete. That's a powerful thing to have. That's what we want to test with the chopped up 3 in the image set above.

We'll need to create smaller versions of these PNG images rescaled to 28 by 28 pixels, to match what we've used from the MNIST data. You can use your image editor to do this.

Python libraries again help us out with reading and decoding the data from common image file formats, including the PNG format. Have a look at the following simple code:

```
import scipy.misc
img_array = scipy.misc.imread(image_file_name, flatten=True)
```

```
img_data  = 255.0 - img_array.reshape(784)
img_data = (img_data / 255.0 * 0.99) + 0.01
```

The scipy.misc.imread() function is the one that helps us get data out of image files such as PNG or JPG files. We have to import the scipy.misc library to use it. The “flatten=True” parameter turns the image into simple array of floating point numbers, and if the image were coloured, the colour values would be flattened into grey scale, which is what we need.

The next line reshapes the array from a 28x28 square into a long list of values, which is what we need to feed to our neural network. We’ve done that many times before. What is new is the subtraction of the array’s values from 255.0. The reason for this is that it is conventional for 0 to mean black and 255 to mean white, but the MNIST data set has this the opposite way around, so we have to reverse the values to match what the MNIST data does.

The last line does the familiar rescaling of the data values so they range from 0.01 to 1.0.

Sample code to demonstrate the reading of PNG files is always online at gihub:

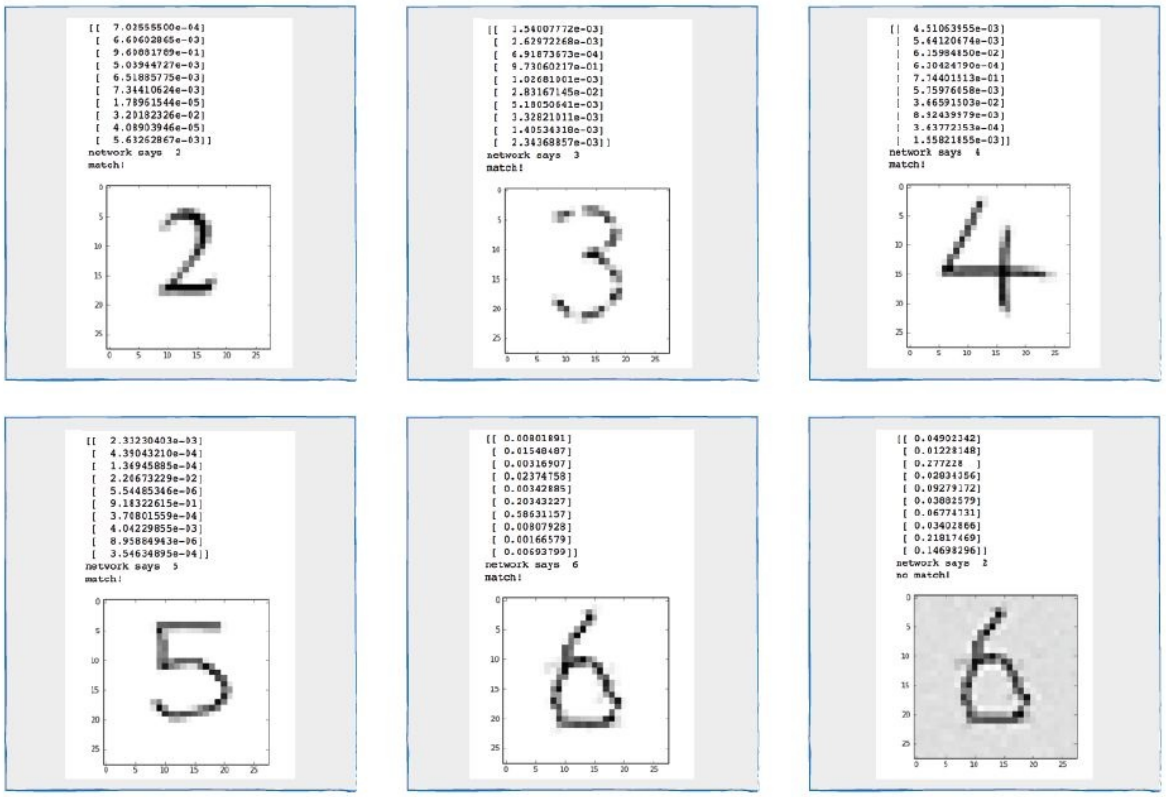
- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

We need to create a version of our basic neural network program that trains on the MNIST training data set but instead of testing with the MNIST test set, it tests against data created from our own images.

The new program is online at github:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

Does it work? It does! The following summarises the results of querying with our own images.



You can see that the neural network recognised all of the images we created, including the deliberately damaged “3”. Only the “6” with added noise failed.

Try it with your own images, especially handwritten, to prove to yourself that the neural network really does work.

And see how far you can go with damaged or deformed images. You’ll be impressed with how resilient the neural network is.

Inside the Mind of a Neural Network

Neural networks are useful for solving the kinds of problems that we don't really know how to solve with simple crisp rules. Imagine writing a set of rules to apply to images of handwritten numbers in order to decide what the number was. You can imagine that wouldn't be easy, and our attempts probably not very successful either.

Mysterious Black Box

Once a neural network is trained, and performs well enough on test data, you essentially have a mysterious **black box**. You don't really know **how** it works out the answer - it just does.

This isn't always a problem if you're just interested in answers, and don't really care how they're arrived at. But it is a disadvantage of these kinds of machine learning methods - the learning doesn't often translate into understanding or wisdom about the problem the black box has learned to solve.

Let's see if we can take a peek inside our simple neural network to see if we can understand what it has learned, to visualise the knowledge it has gathered through training.

We could look at the weights, which is after all what the neural network learns. But that's not likely to be that informative. Especially as the way neural networks work is to distribute their learning across different link weights. This gives them an advantage in that they are resilient to damage, just like biological brains are. It's unlikely that removing one node, or even quite a few nodes, will completely damage the ability of a neural network to work well.

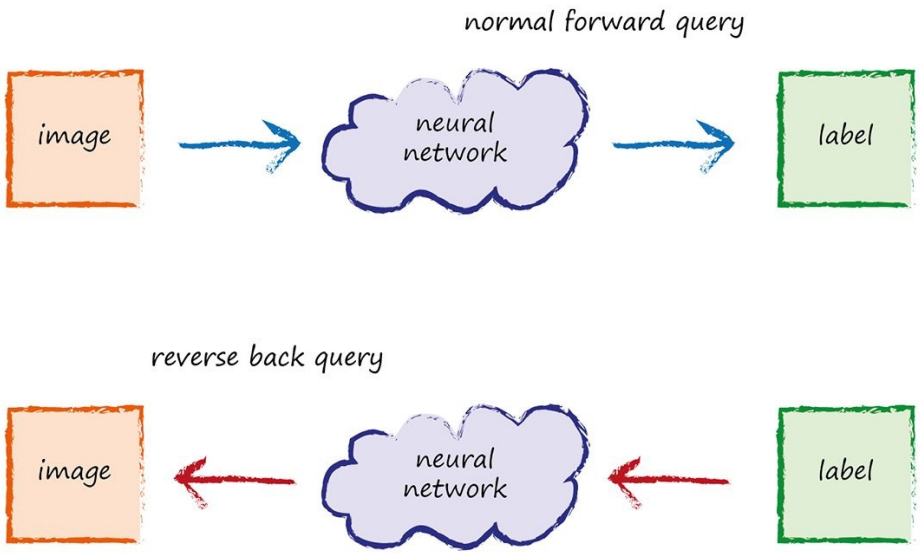
Here's a crazy idea.

Backwards Query

Normally, we feed a trained neural network a question, and out pops an answer. In our example, that question is an image of a human handwritten number. The answer is a label representing a number from 0 to 9.

What if we turned this around and did it backwards? What if we fed a label into the output nodes, and fed the signal backwards through the already-trained network, until out popped an image from the input nodes? The following

diagram shows the normal forward query, and this crazy reverse **back query** idea.



We already know how to propagate signals through a network, moderating them with link weights, and recombining them at nodes before applying an activation function. All this works for signals flowing backwards too, except that the inverse activation is used. If **y = f(x)** was the forward activation then the inverse is **x = g(y)**. This isn't that hard to work out for the logistic function using simple algebra:

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

$$1 + e^{-x} = 1/y$$

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

$$-x = \ln [(1-y) / y]$$

$$x = \ln [y / (1-y)]$$

This is called the **logit** function, and the Python scipy.special library provides this function as **scipy.special.logit()**, just like it provides **scipy.special.expit()** for the logistic sigmoid function.

Before applying the logit() inverse activation function, we need to make sure the signals are valid. What does this mean? Well, you remember the logistic sigmoid function takes any value and outputs a value somewhere between 0 and 1, but

not including 0 and 1 themselves. The inverse function must take values from the same range, somewhere between 0 and 1, excluding 0 and 1, and pop out a value that could be any positive or negative number. To achieve this, we simply take all the values at a layer about to have the logit() applied, and rescale them to the valid range. I've chosen the range 0.01 to 0.99.

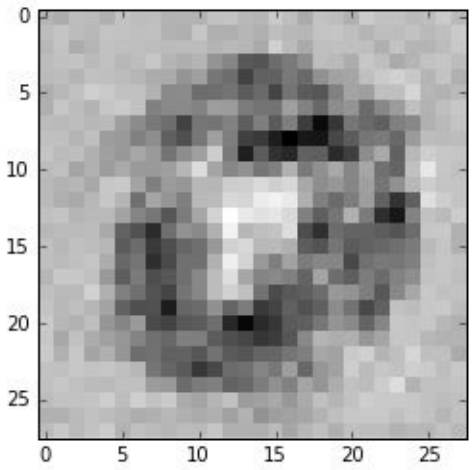
The code is always available online at github at the following link:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

The Label "0"

Let's see what happens if we do a back query with the label "0". That is, we present values to the output nodes which are all 0.01 except for the first node representing the label "0" where we use the value 0.99. In other words, the array **[0.99, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]**.

The following shows the image that pops out of the input nodes.



That is interesting!

That image is a privileged insight into the mind of a neural network. What does it mean? How do we interpret it?

The main thing we notice is that there is a round shape in the image. That makes sense, because we're asking the neural network what the ideal question is for an answer to be "0".

We also notice dark, light and some medium grey areas:

- The **dark** areas are the parts of the question image which, if marked by a pen, make up supporting evidence that the answer should be a “0”. These make sense as they seem to form the outline of a zero shape.

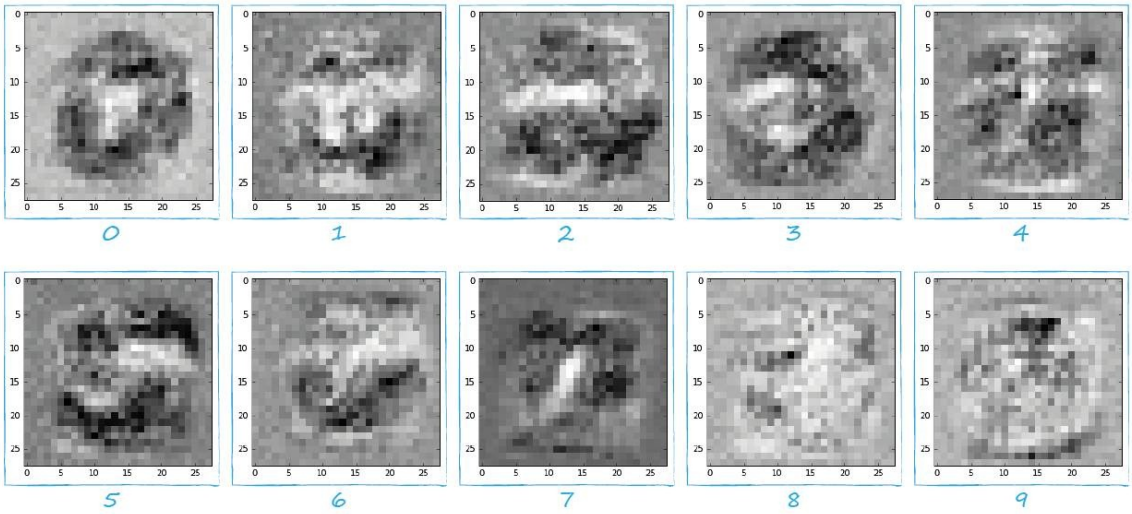
- The **light** areas are the bits of the question image which should remain clear of any pen marks to support the case that the answer is a “0”. Again these make sense as they form the middle part of a zero shape.

- The neural network is broadly indifferent to the **grey** areas.

So we have actually understood, in a rough sense, what the neural network has learned about classifying images with the label “0”.

That’s a rare insight as more complex networks with more layers, or more complex problems, may not have such readily interpretable results. You’re encouraged to experiment and gave a go yourself!

More Brain Scans
The following shows the results of back querying the rest of the digits.



Wow! Again some really interesting images. They're like ultrasound scans into the brain of the neural network.

Some notes about these images:

- The "7" is really clear. You can see the dark bits which, if marked in the query image, strongly suggest a label "7". You can also see the additional "white" area which must be clear of any marking. Together these two characteristics indicate a "7".

- The same applies to the "3" - there are dark areas which, if marked, indicate a "3", and there are white areas which must be clear.
- The “2” and "5" are similarly clear too.
- The "4" is interesting in that there is a shape which appears to have 4 quadrants, and excluded areas too.
- The "8" is largely made up of a “snowman” shaped white areas suggesting that an eight is characterised by markings kept out of these “head and body” regions.
- The “1” is rather puzzling. It seems to focus more on areas which much be kept clear than on areas which must be marked. That’s ok, it’s what the network happens to have learned from the examples.
- The “9” is not very clear at all. It does have a definite dark area and some finer shapes for the white areas. This is what the network has learned, and overall, when combined with what it has learned for the rest of the digits, allows the network to perform really well at 97.5% accuracy. We might look at this image and conclude that more training examples might help the network learn a clearer template for “9”.

So there you have it - a rare insight into the workings of the mind of a neural network.

Creating New Training Data: Rotations

If you think about the MNIST training data you realise that it is quite a rich set of examples of how people write numbers. There are all sorts of styles of handwriting in there, good and bad too.

The neural network has to learn as many of these variations as possible. It does help that there are many forms of the number "4" in there. Some are squished, some are wide, some are rotated, some have an open top and some are closed.

Wouldn't it be useful if we could create yet more such variations as examples? How would we do that? We can't easy collect thousands more examples of human handwriting. We could but it would be very laborious.

A cool idea is to take the existing examples, and create new ones from those by rotating them clockwise and anticlockwise, by 10 degrees for example. For each training example we could have two additional examples. We could create many more examples with different rotation angles, but for now let's just try +10 and -10 degrees to see if the idea works.

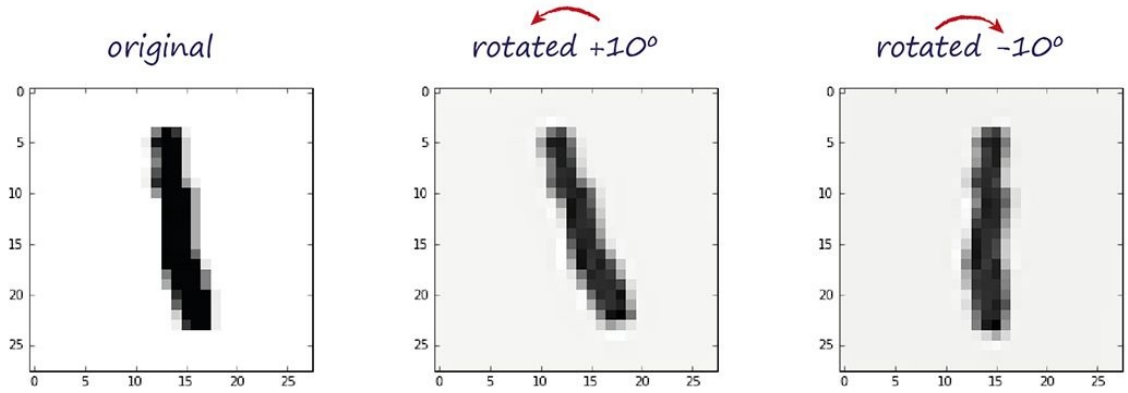
Python's many extensions and libraries come to the rescue again. The ndimage.interpolation.rotate() can rotate an array by a given angle, which is exactly what we need. Remember that our inputs are a one-dimensional long list of length 784, because we've designed our neural networks to take a long list of input signals. We'll need to reshape that long list into a 28*28 array so we can rotate it, and then unroll the result back into a 784 long list of input signals before we feed it to our neural network.

The following code shows how we use the ndimage.interpolation.rotate() function, assuming we have the **scaled_input** array from before:

```
# create rotated variations
# rotated anticlockwise by 10 degrees
inputs_plus10_img = scipy.ndimage.interpolation.rotate(scaled_input.reshape(28,28), 10, cval=0.01, reshape=False)
# rotated clockwise by 10 degrees
inputs_minus10_img = scipy.ndimage.interpolation.rotate(scaled_input.reshape(28,28), -10, cval=0.01, reshape=False)
```

You can see that the original scaled_input array is reshaped to a 28 by 28 array, then scaled. That reshape=False parameter prevents the library from being overly helpful and squishing the image so that it all fits after the rotation without any bits being clipped off. The cval is the value used to fill in array elements because they didn't exist in the original image but have now come into view. We don't want the default value of 0.0 but instead 0.01 because we've shifted the range to avoid zeros being input to our neural network.

Record 6 (the seventh record) of the smaller MNIST training set is a handwritten number "1". You can see the original and two additional variations produced by the code in the diagram below.



You can see the benefits clearly. The version of the original image rotated +10 degrees provides an example where someone might have a style of writing that slopes their 1's backwards. Even more interesting is the version of the original rotated -10 degrees, which is clockwise. You can see that this version is actually straighter than the original, and in some sense a more representative image to learn from.

Let's create a new Python notebook with the original neural network code, but now with additional training examples created by rotating the originals 10 degrees in both directions. This code is available online at github at the following link:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

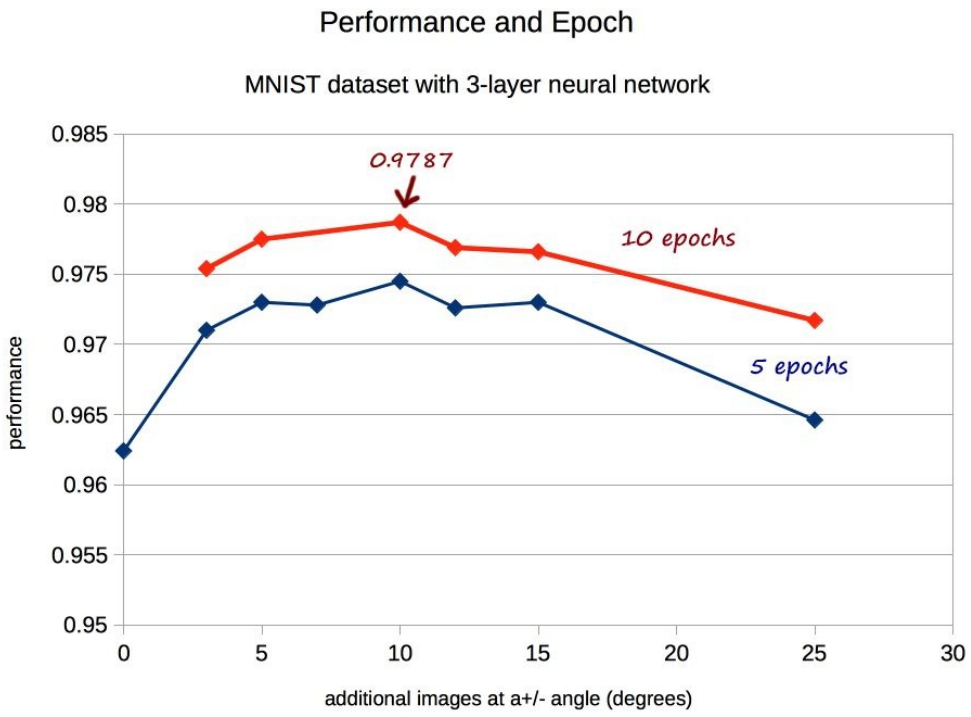
An initial run with learning rate set to 0.1, and only one training epoch, the resultant performance was **0.9669**. That's a solid improvement over 0.954

without the additional rotated training images. This performance is already amongst the better ones listed on Yann LeCunn's website.

Let's run a series of experiments, varying the number of epochs to see if we can push this already good performance up even more. Let's also reduce the **learning rate to 0.01** because we are now creating much more training data, so can afford to take smaller more cautious learning steps, as we've extended the learning time overall.

Remember that we don't expect to get 100% as there is very likely an inherent limit due to our specific neural network architecture or the completeness of our training data, so we may never hope to get above 98% or some other figure. By "specific neural network architecture" we mean the choice of nodes in each layer, the choice of hidden layers, the choice of activation function, and so on.

Here's the graph showing the performance as we vary the angle of additional rotated training images. The performance without the additional rotated training examples is also shown for easy comparison.



You can see that with 5 epochs the best result is **0.9745** or 97.5% accuracy. That is a jump up again on our previous record.

It is worth noticing that for large angles the performance degrades. That makes sense, as large angles means we create images which don't actually represent the numbers at all. Imagine a "3" on its side at 90 degrees. That's not a three anymore. So by adding training examples with overly rotated images, we're reducing the quality of the training by adding false examples. Ten degrees seems to be the optimal angle for maximising the value of additional data.

The performance for 10 epochs peaks at a record breaking **0.9787** or almost **98%**! That is really a stunning result, amongst the best for this kind of simple network. Remember we haven't done any fancy tricks to the network or data that some people will do, we've kept it simple, and still achieved a result to be very proud of.

Well done!

Epilogue

In this guide I hope that you have seen how some problems are easy for humans to solve, but hard for traditional computer approaches. Image recognition is one of these so-called “artificial intelligence” challenges.

Neural networks have enabled huge progress on image recognition, and a wide range of other kinds of hard problems too. A key part of their early motivation was the puzzle that biological brains - like pigeon or insect brains - appeared to be simpler and slower, than today’s huge supercomputers and yet they could carry out complex tasks like flight, feeding and building homes. Those biological brains also seemed extremely resilient to damage, or to imperfect signals. Digital computers and traditional computing weren’t either of these things.

Today, neural networks are a key part of some of the most fantastic successes in artificial intelligence. There is continued huge interest in neural networks and machine learning, especially **deep learning** - where a hierarchy of machine learning methods are used. In early 2016, Google’s DeepMind beat a world master at the ancient game of Go. This is a massive milestone for artificial intelligence, because Go requires much deeper strategy and nuance than chess, for example, and researchers had thought a computer playing that well was years off. Neural networks played a key role in that success.

I hope you’ve seen how the core ideas behind neural networks are actually quite simple. And I hope you’ve had fun experimenting with neural networks too. Perhaps you’ve developed an interest to explore other kinds of machine learning and artificial intelligence.

If you’ve done any of these things, I’ve succeeded.

Appendix A: A Gentle Introduction to Calculus

Imagine you're in a car cruising smoothly and relaxed at a constant 30 miles per hour. Imagine you then press the accelerator pedal. If you keep it pressed your speed increases to 35, 40, 50, and 60 miles per hour.

The speed of the car **changes**.

In this section we'll explore the idea of things changing - like the speed of a car - and how to work out that change mathematically. What do we mean, mathematically? We mean understanding how things are related to each other, so we can work out precisely how changes in one result in changes in another. Like car speed changing with the time on my watch. Or plant height changing with rain levels. Or the extension of a metal spring changing as we apply different amounts of pulling force.

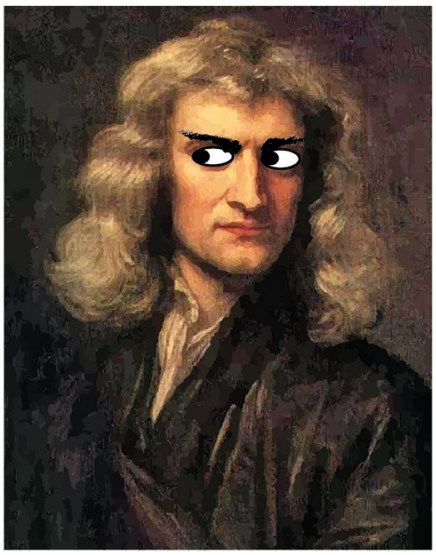
This is called **calculus** by mathematicians. I hesitated about calling this section calculus because many people seem to think that is a hard scary subject to be avoided. That's a real shame, and the fault of bad teaching and terrible school textbooks.

By the end of this appendix, you'll see that working out how things change in a mathematically precise way - because that's all calculus really is - is not that difficult for many useful scenarios.

Even if you've done calculus or **differentiation** already, perhaps at school, it is worth going through this section because we will understand how calculus was invented back in history. The ideas and tools used by those pioneering mathematicians are really useful to have in your back pocket, and can be very helpful when trying to solve different kinds of problems in future.

If you enjoy a good historical punch-up, look up the drama between Leibniz and Newton who both claimed to have invested calculus first!

Gottfried Leibniz



Sir Isaac Newton

A Flat Line

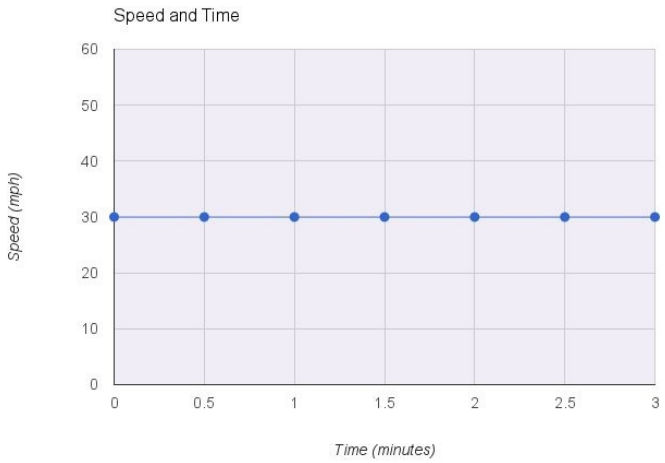
Let’s first start with a very easy scenario to get ourselves settled and ready to go.

Imagine that car again, but cruising at a constant speed of 30 miles per hour. Not faster, not slower, just 30 miles per hour.

Here’s a table showing the speed at various points in time, measured every half a minute.

Time (mins)	Speed (mph)
0.0	30
0.5	30
1.0	30
1.5	30
2.0	30
2.5	30
3.0	30

The following graph visualises this speed at those several points in time.



You can see that the speed doesn't change over time, that's why it is a straight horizontal line. It doesn't go up (faster) or down (slower), it just stays at 30 miles per hour.

The mathematical expression for the speed, which we'll call **s**, is

$$s = 30$$

Now, if someone asked how the speed changes with time, we'd say it didn't. The rate of change is zero. In other words, the speed doesn't depend on time. That dependency is zero.

We've just done calculus! We have, really!

Calculus is about establishing how things change as a result of other things changing. Here we are thinking about **how speed changes with time**.

There is a mathematical way of writing this.

$$\frac{\delta s}{\delta t} = 0$$

What are those symbols? Think of the symbols meaning "how speed changes when time changes", or "how does s depend on **t**".

So that expression is a mathematician's concise way of saying the speed doesn't change with time. Or put another way, the passing of time doesn't affect speed. The dependency of speed on time is zero. That's what the zero in the expression means. They are completely independent. Ok, ok - we get it!

In fact you can see this independence when you look again at the expression for the speed, **s** = 30. There is no mention of the time in there at all. That is, there is no symbol **t** hidden in that expression. So we don't need to do any fancy calculus to work out that $\partial \mathbf{s} / \partial \mathbf{t} = 0$, we can do it by simply looking at the expression. Mathematicians call this "by inspection".

Expressions like $\partial \mathbf{s} / \partial \mathbf{t}$, which explain a rate of change, are called **derivatives**. We don't need to know this for our purposes, but you may come across that word elsewhere.

Now let's see what happens if we press the accelerator. Exciting!

A Sloped Straight Line

Imagine that same car going at 30 miles per hour. We press the accelerator gently and the car speeds up. We keep it pressed and watch the speed dial on the dashboard and note the speed every 30 seconds.

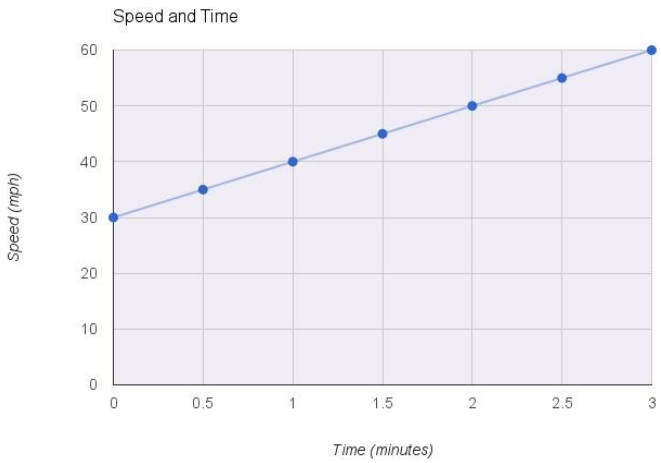
After 30 seconds the car is going at 35 miles per hour. After 1 minute the car is now going at 40 miles per hour. After 90 seconds it's going at 45 miles per hour, and after 2 minutes it's 50 miles per hour. The car keeps speeding up by 10 miles per hour every minute.

Here's the same information summarised in a table.

Time (mins)	Speed (mph)
0.0	30
0.5	35
1.0	40
1.5	45
2.0	50

2.5	55
3.0	60

Let’s visualise this again.



You can see that the speed increases from 30 miles per hour all the way up to 60 miles per hour at a **constant rate**. You can see this is a constant rate because the increments in speed are the same every half a minute and this leads to a straight line for speed.

What’s the expression for the speed? Well we must have speed 30 at time zero. And after that we add an extra 10 miles per hour every minute. So the expression is as follows.

$$speed = 30 + (10 * time)$$

Or using symbols,

$$s = 30 + 10t$$

You can see the constant 30 in there. And you can also see the (10 * time) which adds the 10 miles per hour for every minute. You’ll quickly realise that the 10 is the **gradient** of that line we plotted. Remember the general form of straight lines is **y** = **ax** + **b** where a is the **slope**, or **gradient**.

So what's the expression for how speed changes with time? Well, we've already been talking about it, speed increases 10 mph every minute.

$$\frac{\delta s}{\delta t} = 10$$

What this is saying, is that there is indeed a dependency between speed and time. This is because $\partial \mathbf{s} / \partial \mathbf{t}$ is not zero.

Remembering that the slope of a straight line $\mathbf{y} = \mathbf{ax} + \mathbf{b}$ is **a**, we can see "by inspection" that the slope of $\mathbf{s} = 30 + 10\mathbf{t}$ will be 10.

Great work! We've covered a lot of the basics of calculus already, and it wasn't that hard at all.

Now let's hit that accelerator harder!

A Curved Line

Imagine I started the car from stationary and hit the accelerator hard and kept it pressed hard. Clearly the starting speed is zero because we're not moving initially.

Imagine we're pressing the accelerator so hard that the car doesn't increase it's speed at a constant rate. Instead it increases it's speed in faster way. That means, it is not adding 10 miles per hour every minute. Instead it is adding speed at a rate that itself goes up the longer we keep that accelerator pressed.

For this example, let's imagine the speed is measured every minute as shown in this table.

Time (mins)	Speed (mph)
0	0
1	1
2	4
3	9

4	16
5	25
6	36
7	49
8	64

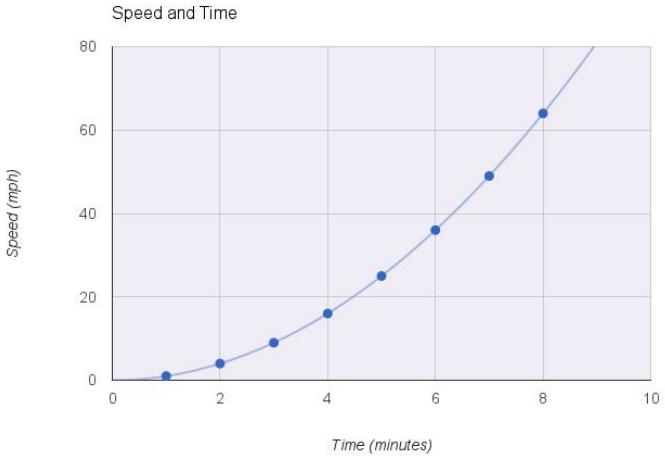
If you look closely you might have spotted that I've chosen to have the speed as the square of the time in minutes. That is, the speed at time 2 is $2^2=4$, and at time 3 is $3^2=9$, and time 4 is $4^2=16$, and so on.

The expression for this is easy to write too.

$$s = t^2$$

Yes, I know this is a very contrived example of car speed, but it will illustrate really well how we might do calculus.

Let's visualise this so we can get a feel for how the speed changes with time.



You can see the speed zooming upwards at an ever faster rate. The graph isn't a straight line now. You can imagine that the speed would explode to very high numbers quite quickly. At 20 minutes the speed would be 400 miles per hour. And at 100 minutes the speed would be 10,000 miles per hour!

The interesting question is - what's the rate of change for the speed with respect to time? That is, how does the speed change with time?

This isn't the same as asking, what is the actual speed at a particular time. We already know that because we have the expression $\mathbf{s} = \mathbf{t}^2$.

What we're asking is this - at any point in time, what is the **rate of change** of speed? What does this even mean in this example where the graph is curved?

If we think back to our previous two examples, we saw that the rate of change was the slope of the graph of speed against time. When the car was cruising at a constant 30, the speed wasn't changing, so it's rate of change was zero. When the car was getting faster steadily, it's rate of change was 10 miles per hour every minute. And that 10 mph every minute was true for any point in time. At time 2 minutes the rate of change was 10 mph/min. And it was true at 4 minutes and would be true at 100 minutes too.

Can we apply this same thinking to this curved graph? Yes we can - but let's go extra slowly here.

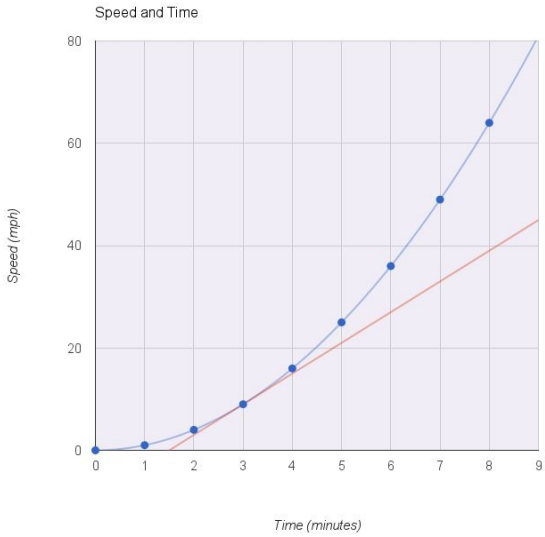
Calculus By Hand

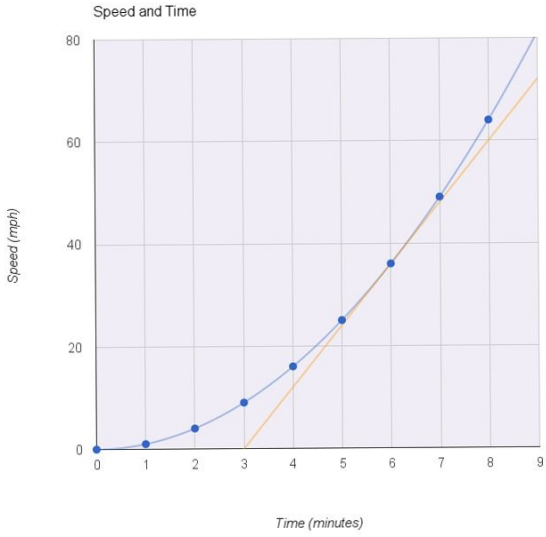
Let's look more closely at what is happening at time 3 minutes.

At 3 minutes, the speed is 9 miles per hour. We know it will be faster after 3 minutes. Let's compare that with what is happening at 6 minutes. At 6 minutes, the speed is 36 miles per hour. We also know that the speed will be faster after 6 minutes.

But we also know that a moment after 6 minutes the speed increase will be greater than an equivalent moment after 3 minutes. There is a real difference between what's happening at time 3 minutes and time 6 minutes.

Let's visualise this as follows.





You can see that the slope at time 6 minutes is steeper than at time 3 minutes. These slopes are the rate of change we want. This is an important realisation, so let's say it again. The rate of change of a curve at any point, is the slope of the curve at that point.

But how do we measure the slope of a line that is curved? Straight lines were easy, but curvy lines? We could try to **estimate** the slope by drawing a straight line, called a **tangent**, which just touches that curved line in a way that tries to be at the same gradient as the curve just at the point. This is in fact how people used to do it before other ways were invented.

Let's try this rough and ready method, just so we have some sympathy with that approach. The following diagram shows the speed graph with a tangent touching the speed curve at time 6 minutes.

To work out the slope, or gradient, we know from school maths that we need to divide the height of the incline by the extent. In the diagram this height (speed) is shown as **Δs**, and the extent (time) is shown as **Δt**. That symbol, **Δ** called "delta", just means a small change. So **Δt** is a small change in **t**.

The slope is **Δs / Δt**. We could chose any sized triangle for the incline, and use a ruler to measure the height and extent. With my measurements I just happen to have a triangle with **Δs** measured as 9.6, and **Δt** as 0.8. That gives the slope as follows:

$$\text{rate of change at a point} = \text{slope at that point}$$

$$= \frac{\Delta s}{\Delta t}$$

$$= 9.6 / 0.8$$

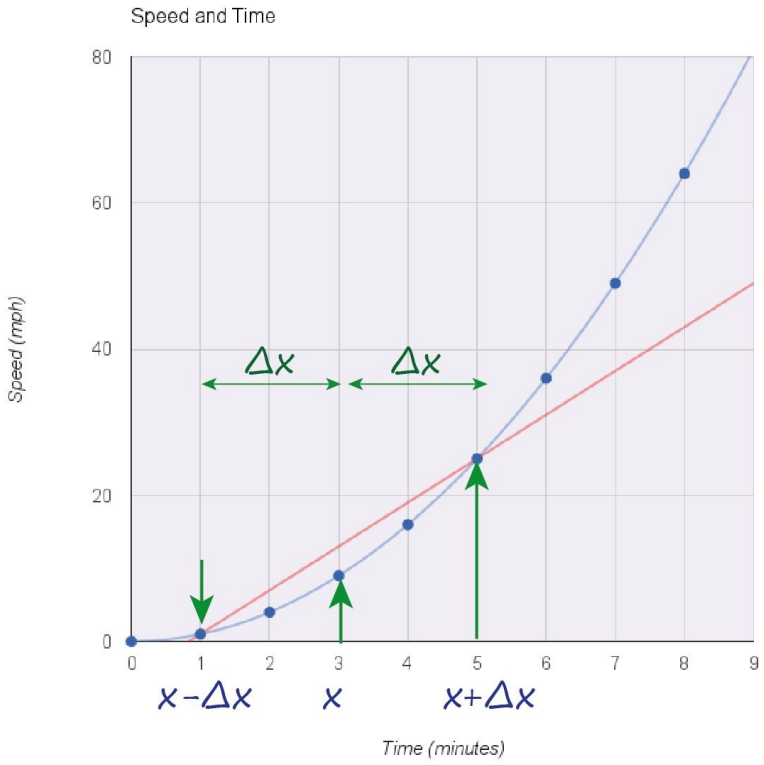
$$= 12.0$$

We have a key result! The rate of change of speed at time 6 minutes is 12.0 mph per min.

You can see that relying on a ruler, and even trying to place a tangent by hand, isn't going to be very accurate. So let's get a tiny bit more sophisticated.

Calculus Not By Hand

Look at the following graph which has a new line marked on it. It isn't the tangent because it doesn't touch the curve only at a single point. But is does seem to be centred around time 3 minutes in some way.



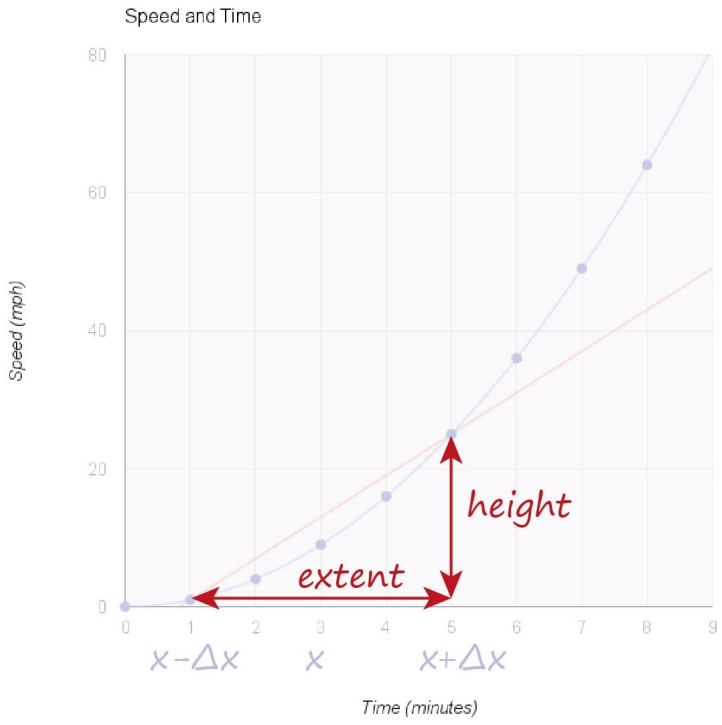
In fact there is connection to time 3 minutes. What we've done is chosen a time above and below this point of interest at **t**=3. Here, we've selected points 2 minutes above and below **t**=3 minutes. That is, **t**=1 and **t**=5 minutes.

Using our mathematical notation, we say we have a **Δx** of 2 minutes. And we have chosen points **x-Δx** and **x+Δx**. Remember that symbol **Δ** just means a "small change", so **Δx** is a small change in **x**.

Why have we done this? It'll become clear very soon - hang in there just a bit.

If we look at the speeds at times **x-Δx** and **x+Δx**, and draw a line between between those two points, we have something that very roughly has the same slope as a tangent at the middle point **x**. Have a look again at the diagram above to see that straight line. Sure, it's not going to have exactly the same slope as a true tangent at **x**, but we'll fix this.

Let's work out the gradient of this line. We use the same approach as before where the gradient is the height of the incline divided by the extent. The following diagram makes clearer what the height and extent is here.



The height is the difference between the two speeds at **x-Δx** and **x+Δx**, that is, 1 and 5 minutes. We know the speeds are 1^2=1 and 5^2=25 mph at these points so the difference is 24. The extent is the very simple distance between **x-Δx** and **x+Δx**, that is, between 1 and 5, which is 4. So we have:

$$\text{gradient} = \frac{\text{height}}{\text{extent}}$$

$$= 24 / 4$$

$$= 6$$

The gradient of the line, which is approximates the tangent at **t**=3 minutes, is 6 mph per min.

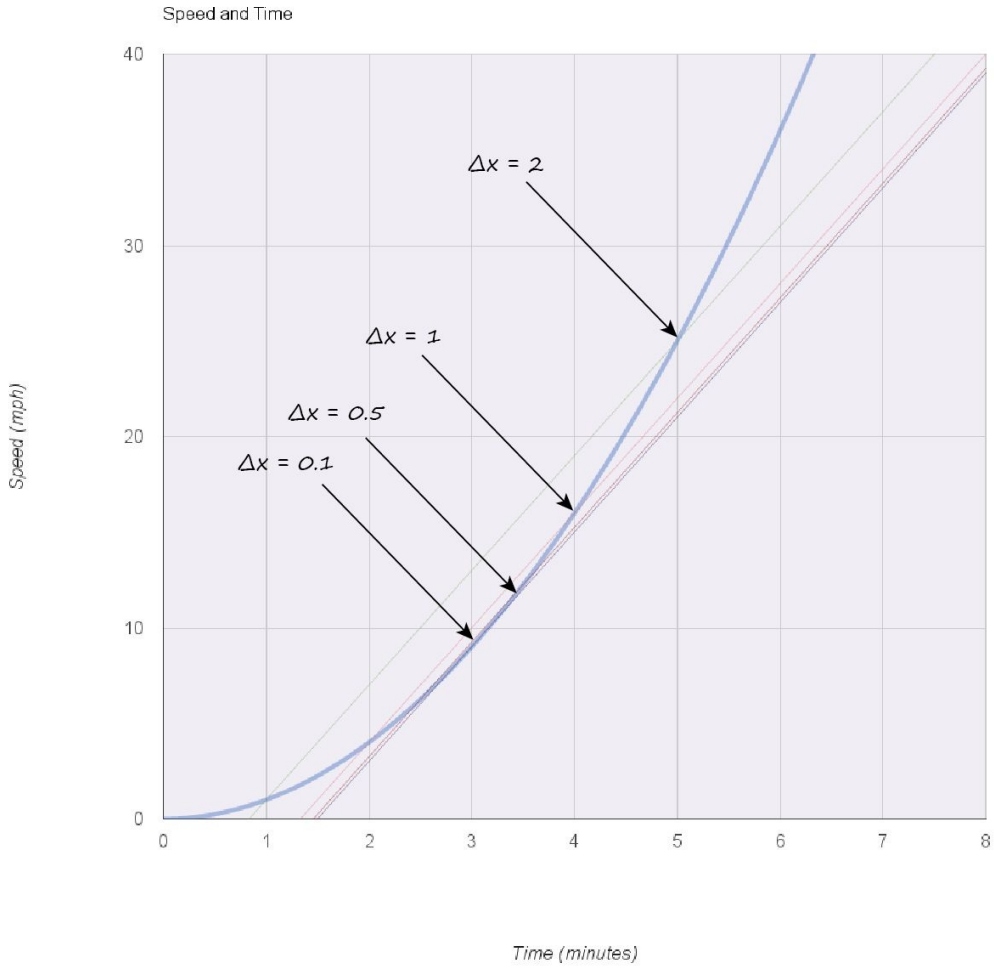
Let's pause and have a think about what we've done. We first tried to work out the slope of a curved line by hand drawing a tangent. This approach will never be accurate, and we can't do it many many times because, being human, we'll get tired, bored, and make mistakes. The next approach doesn't need us to hand draw a tangent, instead we follow a recipe to create a different line which seems to have approximately the right slope. This second approach can be automated by a computer and done many times and very quickly, as no human effort is needed.

That's good but not good enough yet!

That second approach is only an approximation. How can we improve it so it's not an approximation? That's our aim after all, to be able to work out how things change, the gradient, in a mathematically precise way.

This is where the magic happens! We'll see one of the neat tools that mathematicians have developed and had a bit too much fun with!

What would happen if we made the extent smaller? Another way of saying that is, what would happen if we made the **Δx** smaller? The following diagram illustrates several approximations or slope lines resulting from a decreasing **Δx**.



We've drawn the lines for **Δx** = 2.0, **Δx** = 1.0, **Δx** = 0.5 and **Δx** = 0.1. You can see that the lines are getting closer to the point of interest at 3 minutes. You can imagine that as we keep making **Δx** smaller and smaller, the line gets closer and closer to a true tangent at 3 minutes.

As **Δx** becomes infinitely small, the line becomes infinitely closer to the true tangent. That's pretty cool!

This idea of approximating a solution and improving it by making the deviations smaller and smaller is very powerful. It allows mathematicians to solve problems which are hard to attack directly. It's a bit like creeping up to a solution from the side, instead of running at it head on!

Calculus without Plotting Graphs

We said earlier, calculus is about understanding how things change in a mathematically precise way. Let's see if we can do that by applying this idea of ever smaller **Δx** to the mathematical expressions that define these things - things like our car speed curves.

To recap, the speed is a function of the time that we know to be **s** = $\mathbf{t^2}$. We want to know how the speed changes as a function of time. We've seen that is the slope of **s** when it is plotted against **t**.

This rate of change ∂**s** / ∂**t** is the height divided by the extent of our constructed lines but where the **Δx** gets infinitely small.

What is the height? It is $(\mathbf{t} + \mathbf{\Delta x})^2 - (\mathbf{t} - \mathbf{\Delta x})^2$ as we saw before. This is just **s** = $\mathbf{t^2}$ where t is a bit below and a bit above the point of interest. That amount of bit is **Δx**.

What is the extent? As we saw before, it is simply the distance between (**t** + **Δx**) and (**t** - **Δx**) which is 2**Δx**.

We're almost there,

$$\frac{\delta s}{\delta t} = \frac{\text{height}}{\text{extent}}$$

$$= \frac{(t + \Delta x)^2 - (t - \Delta x)^2}{2\Delta x}$$

Let's expand and simplify that expression,

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x - t^2 - \Delta x^2 + 2t\Delta x}{2\Delta x}$$

$$= \frac{4t\Delta x}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 2t$$

We've actually been very lucky here because the algebra simplified itself very neatly.

So we've done it! The mathematically precise rate of change is $\partial \mathbf{s} / \partial \mathbf{t} = 2\mathbf{t}$. That means for any time t, we know the rate of change of speed $\partial \mathbf{s} / \partial \mathbf{t} = 2\mathbf{t}$.

At $\mathbf{t} = 3$ minutes we have $\partial \mathbf{s} / \partial \mathbf{t} = 2\mathbf{t} = 6$. We actually confirmed that before using the approximate method. For $\mathbf{t} = 6$ minutes, $\partial \mathbf{s} / \partial \mathbf{t} = 2\mathbf{t} = 12$, which nicely matches what we found before too.

What about $\mathbf{t} = 100$ minutes? $\partial \mathbf{s} / \partial \mathbf{t} = 2\mathbf{t} = 200$ mph per minute. That means after 100 minutes, the car is speeding up at a rate of 200 mph per minute.

Let's take a moment to ponder the magnitude and coolness of what we just did. We have a mathematical expression that allows us to precisely know the rate of change of the car speed at any time. And following our earlier discussion, we can see that changes in s do indeed depend on time.

We were lucky that the algebra simplified nicely, but the simple $\mathbf{s} = \mathbf{t}^2$ didn't give us an opportunity to try reducing the $\mathbf{\Delta x}$ in an intentional way. So let's try another example where the speed of the car is only just a bit more complicated,

$$s = t^2 + 2t$$

$$\frac{\delta s}{\delta t} = \frac{height}{extent}$$

What is the height now? It is the difference between s calculated at $\mathbf{t}+\mathbf{\Delta x}$ and s calculated at $\mathbf{t}-\mathbf{\Delta x}$. That is, the height is $(\mathbf{t} + \mathbf{\Delta x})^2 + 2(\mathbf{t} + \mathbf{\Delta x}) - (\mathbf{t} - \mathbf{\Delta x})^2 - 2(\mathbf{t} - \mathbf{\Delta x})$.

What about the extent? It is simply the distance between $(\mathbf{t} + \mathbf{\Delta x})$ and $(\mathbf{t} - \mathbf{\Delta x})$ which is still $2\mathbf{\Delta x}$.

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)}{2\Delta x}$$

Let's expand and simplify that expression,

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x + 2t + 2\Delta x - t^2 - \Delta x^2 + 2t\Delta x - 2t + 2\Delta x}{2\Delta x}$$

$$= \frac{4t\Delta x + 4\Delta x}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 2t + 2$$

That's a great result! Sadly the algebra again simplified a little too easily. It wasn't wasted effort because there is a pattern emerging here which we'll come back to.

Let's try a another example, which isn't that much more complicated. Let's set the speed of the car to be the cube of the time.

$$s = t^3$$

$$\frac{\delta s}{\delta t} = \frac{height}{extent}$$

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^3 - (t - \Delta x)^3}{2\Delta x}$$

Let's expand and simplify that expression,

$$\frac{\delta s}{\delta t} = \frac{t^3 + 3t^2\Delta x + 3t\Delta x^2 + \Delta x^3 - t^3 + 3t^2\Delta x - 3t\Delta x^2 + \Delta x^3}{2\Delta x}$$

$$= \frac{6t^2\Delta x + 4\Delta x^3}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 3t^2 + \Delta x^2$$

Now this is much more interesting! We have a result which contains a **Δx**, whereas before they were all cancelled out.

Well, remember that the gradient is only correct as the **Δx** gets smaller and smaller, infinitely small.

This is the cool bit! What happens to the **Δx** in the expression $\partial \mathbf{s} / \partial \mathbf{t} = 3\mathbf{t}^2 + \mathbf{\Delta x}^2$ as **Δx** gets smaller and smaller? It disappears! If that sounds surprising, think of a very small value for **Δx**. If you try, you can think of an even smaller one. And an even smaller one .. and you could keep going forever, getting ever closer to zero. So let's just get straight to zero and avoid all that hassle.

That gives is the mathematically precise answer we were looking for:

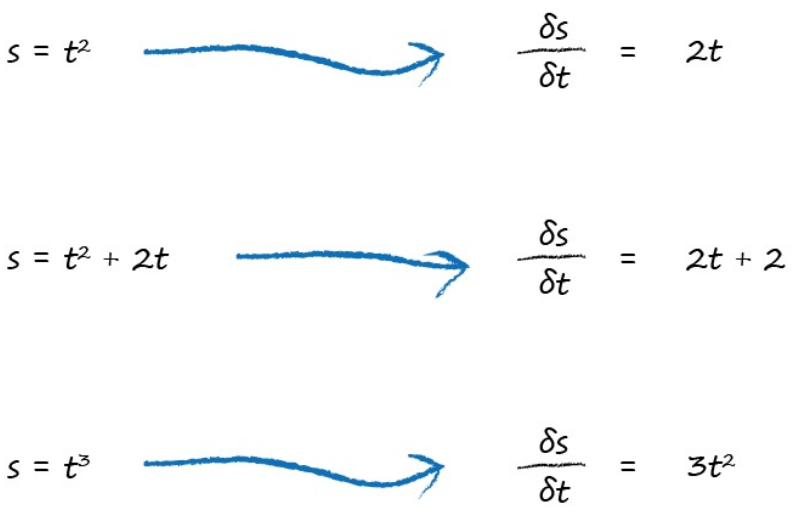
$$\frac{\delta s}{\delta t} = 3t^2$$

That's a fantastic result, and this time we used a powerful mathematical tool to do calculus, and it wasn't that hard at all.

Patterns

As fun as it is to work out derivatives using deltas like **Δx** and seeing what happens when we make them smaller and smaller, we can often do it without doing all that work.

See if you can see any pattern in the derivatives we've worked out so far:



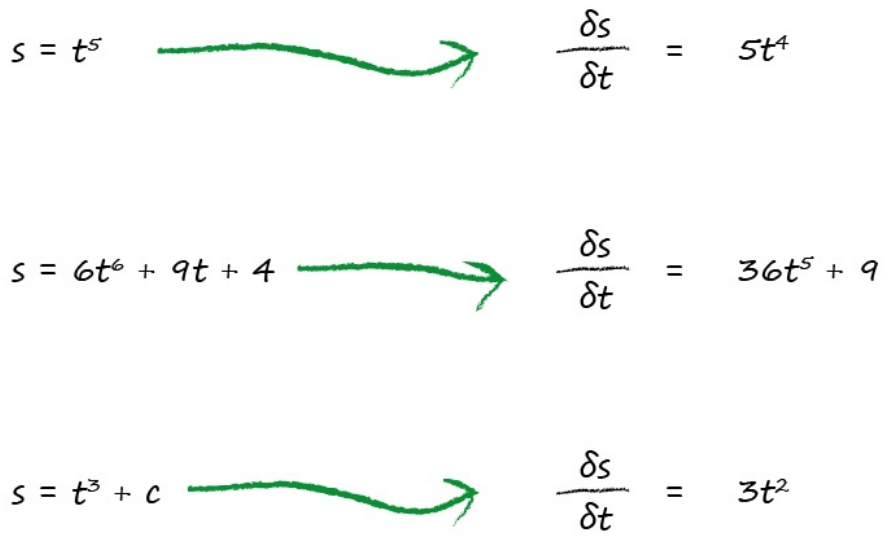
You can see that the derivative of a function of t is the same function but with each power of **t** reduced by one. So $\mathbf{t^4}$ becomes $\mathbf{t^3}$, and $\mathbf{t^7}$ would become $\mathbf{t^6}$, and so on. That's really easy! And if you remember that **t** is just $\mathbf{t^1}$, then in the derivative it becomes $\mathbf{t^0}$, which is 1.

Constant numbers on their own like 3 or 4 or 5 simply disappear. Constant variables on their own, which we might call **a**, **b** or **c**, also disappear, because they too have no rate of change. That's why they're called **constants**.

But hang on, $\mathbf{t^2}$ became 2**t** not just **t**. And $\mathbf{t^3}$ became $3\mathbf{t^2}$ not just $\mathbf{t^2}$. Well there is an extra step where the power is used as a multiplier before it is reduced. So the 5 in $2\mathbf{t^5}$ is used as an additional multiplier before the power is reduced $5*2\mathbf{t^4} = 10\mathbf{t^4}$.

The following summarises this power rule for doing calculus.

Let's try it on some more examples just to practice this new technique.



So this rule allows us to do quite a lot of calculus and for many purposes it's all the calculus we need. Yes, it only works for **polynomials**, that is expressions made of variables with powers like $\mathbf{y} = \mathbf{ax}^{\mathbf{3}} + \mathbf{bx}^{\mathbf{2}} + \mathbf{cx} + \mathbf{d}$, and not with things like sin(**x**) or cos(**x**). That's not a major flaw because there are a huge number of uses for doing calculus with this power rule.

However for neural networks we do need one extra tool, which we'll look at next.

Functions of Functions

Imagine a function

$$f = y^2$$

where y is itself

$$y = x^3 + x$$

We can write this as $\mathbf{f} = (\mathbf{x}^{\mathbf{3}} + \mathbf{x})^2$ if we wanted to.

How does **f** change with **y**? That is, what is $\partial \mathbf{f} / \partial \mathbf{y}$? This is easy as we just apply the power rule we just developed, multiplying and reducing the power, so $\partial \mathbf{f} / \partial \mathbf{y} = 2\mathbf{y}$.

What about a more interesting question - how does **f** change with **x**? Well we could expand out the expression $\mathbf{f} = (\mathbf{x}^3 + \mathbf{x})^2$ and apply this same approach. We can't can't apply it naively to $(\mathbf{x}^3 + \mathbf{x})^2$ to become $2(\mathbf{x}^3 + \mathbf{x})$.

If we worked many of these out the long hard way, using the diminishing deltas approach like before, we'd stumble upon another set of patterns. Let's jump straight to the answer.

The pattern is this:

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} \cdot \frac{\delta y}{\delta x}$$

This is a very powerful result, and is called the **chain rule**.

You can see that it allows us to work out derivatives in layers, like onion rings, unpacking each layer of complexity. To work out $\partial \mathbf{f} / \partial \mathbf{x}$ we might find it easier to work out $\partial \mathbf{f} / \partial \mathbf{y}$ and then also easier to work out $\partial \mathbf{y} / \partial \mathbf{x}$. If these are easier, we can then do calculus on expressions that otherwise look quite impossible. The chain rule allows us to break a problem into smaller easier ones.

Let's look at that example again and apply this chain rule:

$$f = y^2 \quad \text{and} \quad y = x^3 + x$$

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} \cdot \frac{\delta y}{\delta x}$$

We now work out the easier bits. The first bit is (∂**f** / ∂**y**) = 2**y**. The second bit is (∂**y** / ∂**x**) = 3$\mathbf{x}^2$ + 1. So recombining these bits using the chain rule we get

$$\frac{\delta f}{\delta x} = (2y) * (3x^2 + 1)$$

We know that **y** = $\mathbf{x}^3$ + **x** so we can have an expression with only **x**

$$\frac{\delta f}{\delta x} = (2(x^2 + x)) * (3x^2 + 1)$$

$$\frac{\delta f}{\delta x} = (2x^2 + 2x)(3x^2 + 1)$$

Magic!

You may challenge this as ask why we didn't just expand out **f** in terms of **x** first and then apply simple power rule calculus to the resulting polynomial. We could have done that, but we wouldn't have illustrated the chain rule, which allows us to crack much harder problems.

Let's look at just one final example because it shows us how to handle variables which are independent of other variables.

If we have a function

$$f = 2xy + 3x^2z + 4z$$

where **x**, **y** and **z** are independent of each other. What do we mean by independent? We mean that **x**, **y** and **z** can be any value and don't care what the other variables are - they aren't affected by changes in the other variables. This wasn't the case in previous example where **y** was $\mathbf{x}^3$ + **x**, so **y** was dependent on **x**.

What is ∂**f** / ∂**x**? Let's look at each part of that long expression. The first bit is 2**xy**, so the derivative is 2**y**. Why is this so simple? It's simple because **y** is not dependent on **x**. What we're asking when we say ∂**f** / ∂**x** is how does **f** change

when **x** changes. If **y** doesn't depend on **x**, we can treat it like a constant. That **y** might as well be another number like 2 or 3 or 10.

Let's carry on. The next bit is $3\mathbf{x}^{\mathbf{2}}\mathbf{z}$. We can apply the power reduction rule to get 2*3**xz** or 6**xz**. We treat **z** as just a boring constant number like 2 or 4 or maybe 100, because **x** and **z** are independent of each other. A change in **z** doesn't affect **x**.

The final bit 4**z** has no **x** in it at all. So it vanishes completely, because we treat it like a plain constant number like 2 or 4.

The final answer is

$$\frac{\delta f}{\delta x} = 2y + 6xz$$

The important thing in this last example is having the confidence to ignore variables that you know are independent. It makes doing calculus on quite complex expressions drastically simpler, and it is an insight we'll need lots when looking at neural networks.

You can do Calculus!

If you got this far, well done!

You have a genuine insight into what calculus really is, and how it was invented using approximations that get better and better. You can always try these methods on other tough problems that resist normal ways for solving them.

The two techniques we learned, reducing powers and the chain rule, allows us to do quite a lot of calculus, including understanding how neural networks really work and why.

Enjoy your new powers!

Appendix B: Do It with a Raspberry Pi

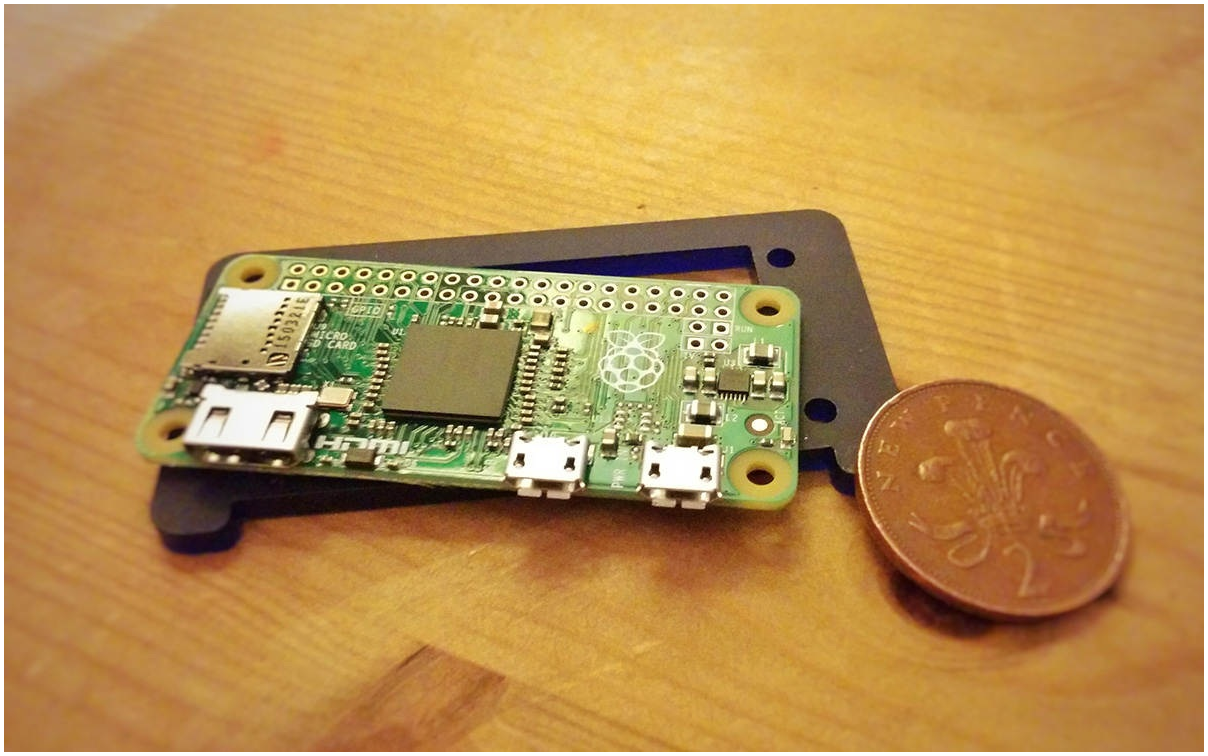
In this section we will aim to get IPython set up on a Raspberry Pi.

There are several good reasons for doing this:

- Raspberry Pis are fairly **inexpensive** and **accessible** to many more people than expensive laptops.
- Raspberry Pis are very open - they run the **free** and **open source** Linux operating system, together with lots of free and open source software, including Python. Open source is important because it is important to understand how things work, to be able to share your work and enable others to build on your work. Education should be about learning how things work, and making your own, and not be about learning to buy closed proprietary software.
- For these and other reasons, they are wildly popular in schools and at home for children who are learning about computing, whether it is software or building hardware projects.
- Raspberry Pis are not as powerful as expensive computers and laptops. So it is an interesting and worthy challenge to be prove that you can still implement a useful neural network with Python on a Raspberry Pi.

I will use a Raspberry Pi Zero because it is even cheaper and smaller than the normal Raspberry Pis, and the challenge to get a neural network running is even more worthy! It costs about £4 UK pounds, or $5 US dollars. That wasn't a typo!

Here's mine, shown next to a 2 penny coin. It's tiny!

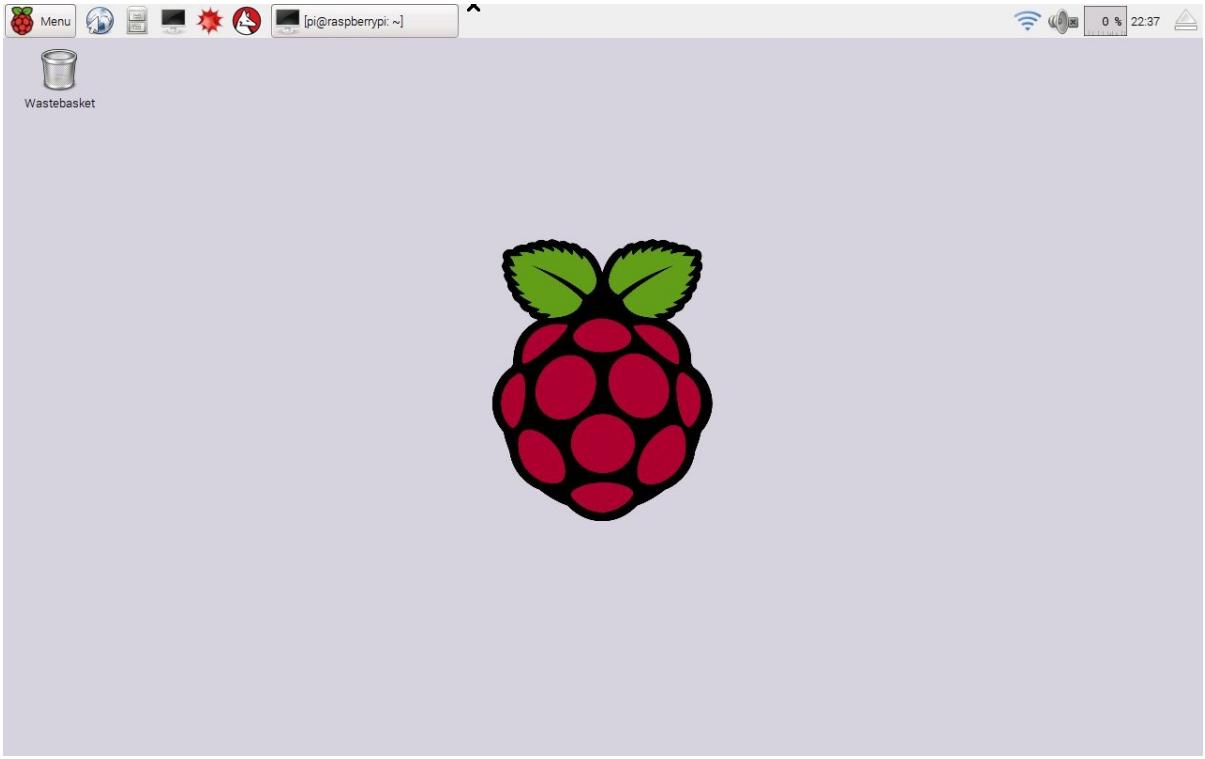


Installing IPython

We'll assume you have a Raspberry Pi powered up and a keyboard, mouse, display and access to the internet working.

There are several options for an operating system, but we'll stick with the most popular which is the officially supported Raspian, a version of the popular Debian Linux distribution designed to work well with Raspberry Pis. Your Raspberry Pi probably came with it already installed. If not install it using the instructions at that link. You can even buy an SD memory card with it already installed, if you're not confident about installing operating systems.

This is the desktop you should see when you start up your Raspberry Pi.

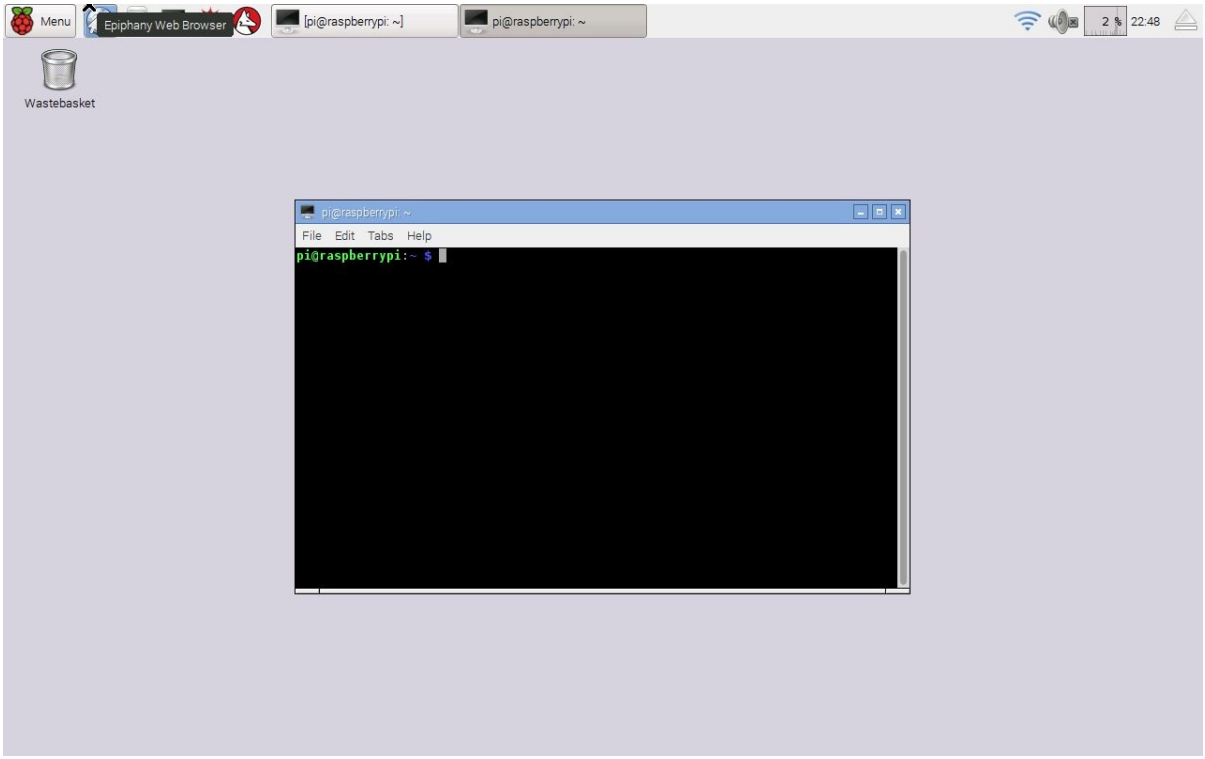


You can see the menu button clearly at the top left, and some shortcuts along the top too.

We're going to install IPython so we can work with the more friendly notebooks through a web browser, and not have to worry about source code files and command lines.

To get IPython we do need to work with the command line, but we only need to do this once, and the recipe is really simple and easy.

Open the Terminal application, which is the icon shortcut at the top which looks like a black monitor. If you hover over it, it'll tell you it is the Terminal. When you run it, you'll be presented with a black box, into which you type commands, looking like the this.



Your Raspberry Pi is very good because it won't allow normal users to issue commands that make deep changes. You have to assume special privileges. Type the following into the terminal:

```
sudo su -
```

You should see the prompt end in with a '#' hash character. It was previously a '$' dollar sign. That shows you now have special privileges and you should be a little careful what you type.

The following commands refresh your Raspberry's list of current software, and then update the ones you've got installed, pulling in any additional software if it's needed.

```
apt-get upgrade
apt-get update
```

Unless you already refreshed your software recently, there will likely be software that needs to be updated. You'll see quite a lot of text fly by. You can safely ignore it. You may be prompted to confirm the update by pressing "y".

Now that our Raspberry is all fresh and up to date, issue the command to get IPython. Note that, at the time of writing, the Raspian software packages don't

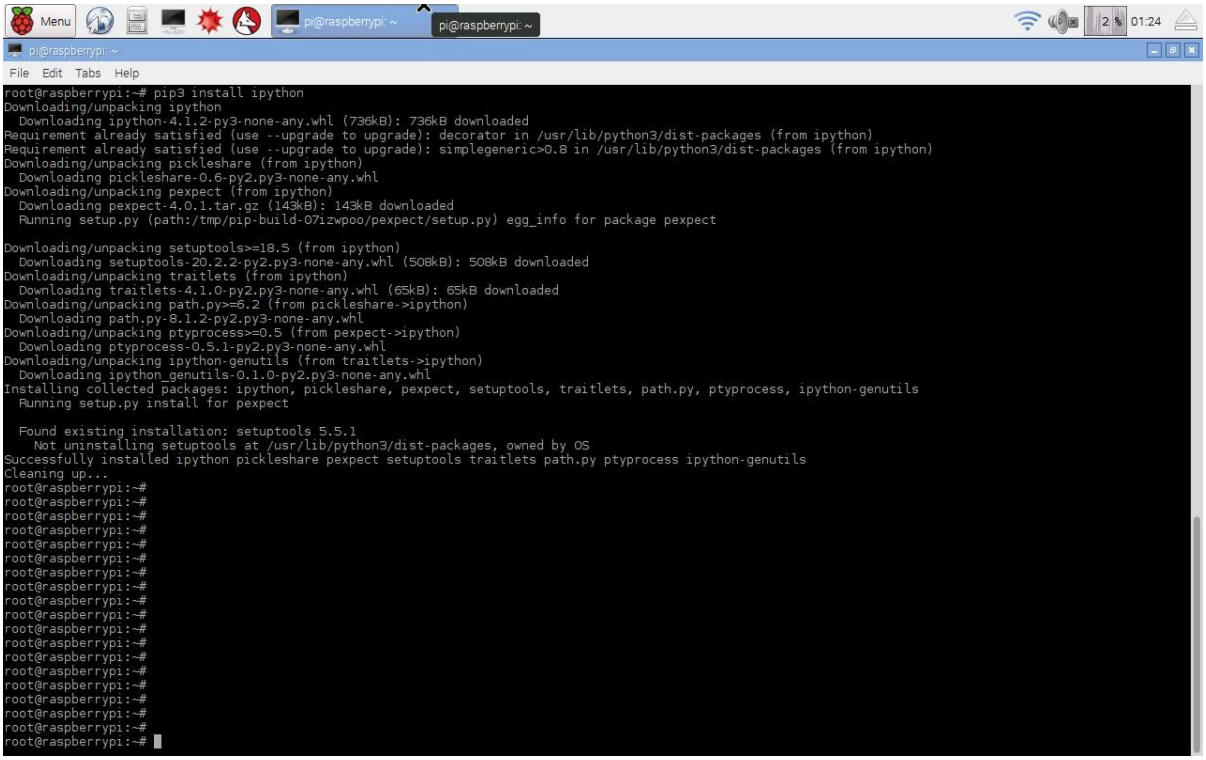
contain a sufficiently recent version of IPython to work with the notebooks we created earlier and put on github for anyone to view and download. If they did, we would simply issue a simple “apt-get install ipython3 ipython3-notebook” or something like that.

If you don’t want to run those notebooks from github, you can happily use the slightly older IPython and notebook versions that come from Raspberry Pi’s software repository.

If we do want to run more recent IPython and notebook software, we need to use some “pip” commands in additional to the “apt-get” to get more recent software from the Python Package Index. The difference is that the software is managed by Python (pip), not by your operating system’s software manager (apt). The following commands should get everything you need.

```
apt-get install python3-matplotlib
apt-get install python3-scipy

pip3 install ipython
pip3 install jupyter
pip3 install matplotlib
pip3 install scipy
```

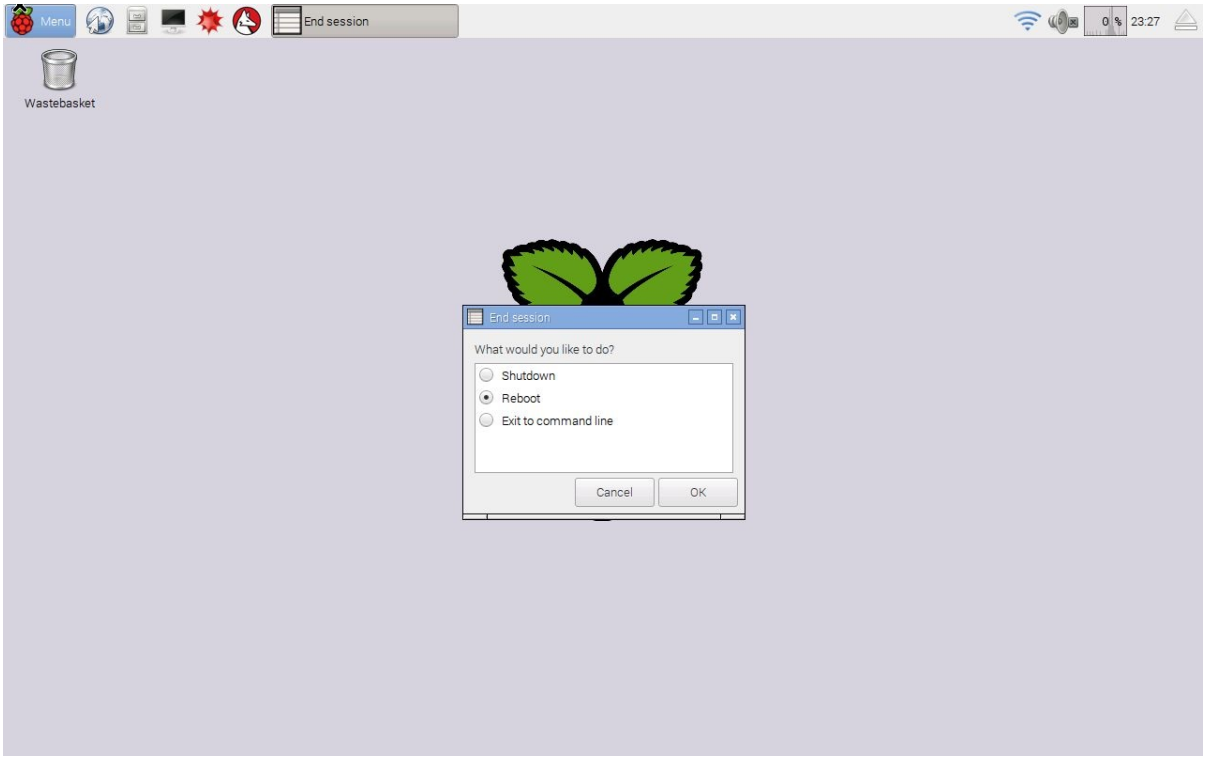
After a bit of text flying by, the job will be done. The speed will depend on your particular Raspberry Pi model, and your internet connection. The following shows my screen when I did this.



The Raspberry Pi normally uses an memory card, called an SD card, just like the ones you might use in your digital camera. They don't have as much space as a normal computer. Issue the following command to clean up the software packages that were downloaded in order to update your Raspberry Pi.

```
apt-get clean
```

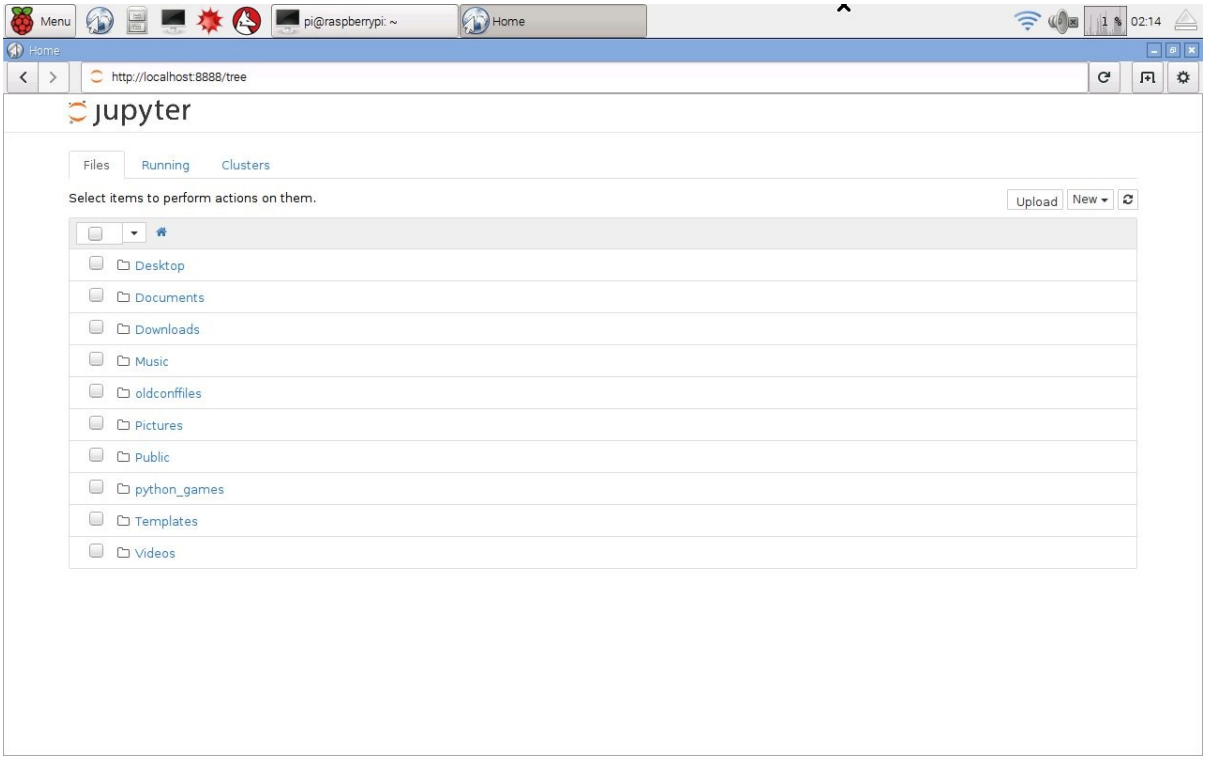
That's it, job done. Restart your Raspberry Pi in case there was a particularly deep change such as a change to the very core of your Raspberry Pi, like a kernel update. You can restart your Raspberry Pi by selecting the "Shutdown …" option from the main menu at the top left, and then choosing "Reboot", as shown next.



After your Raspberry Pi has started up again, start IPython by issuing the following command from the Terminal:

```
jupyter notebook
```

This will automatically launch a web browser with the usual IPython main page, from where you can create new IPython notebooks. Jupyter is the new software for running notebooks. Previously you would have used the “ipython3 notebook” command, which will continue to work for a transition period. The following shows the main IPython starting page.

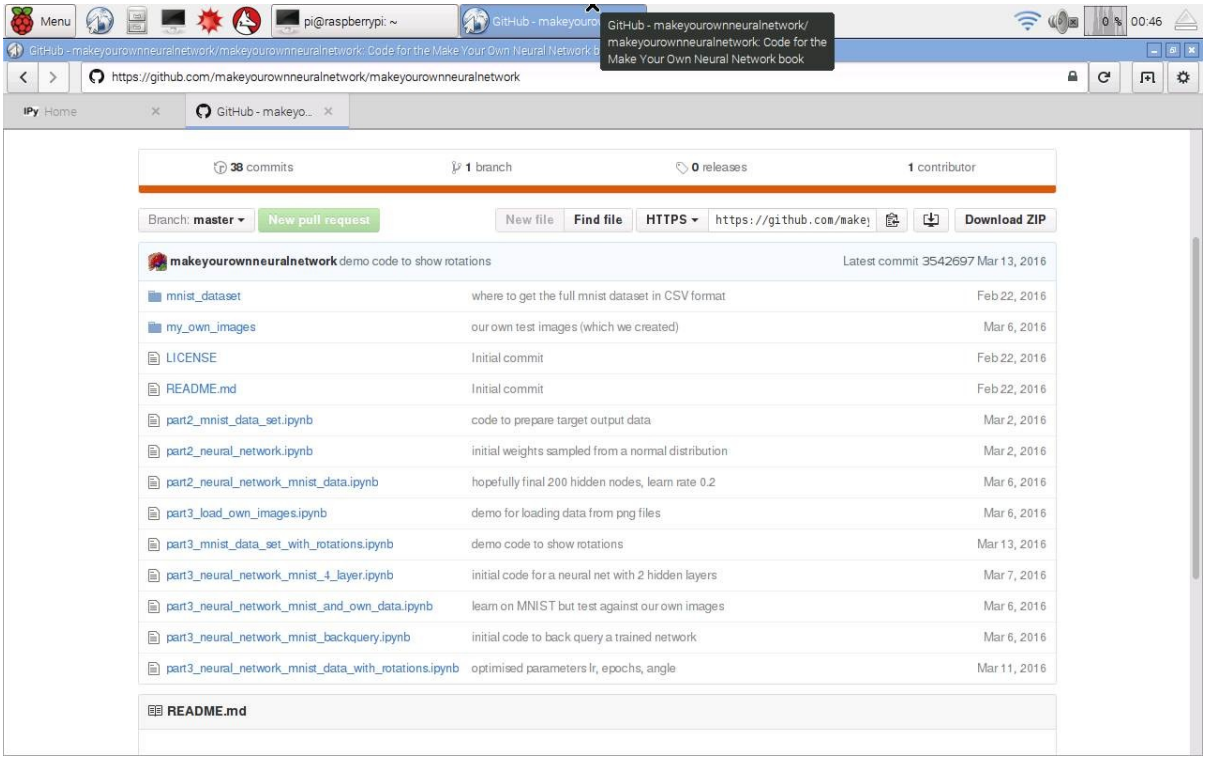


That’s great! So we’ve got IPython up and running on a Raspberry Pi.

You could proceed as normal and create your own IPython notebooks, but we’ll demonstrate that the code we developed in this guide does run. We’ll get the notebooks and also the MNIST dataset of handwritten numbers from github. In a new browser tab go to the link:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwo

You’ll see the github project page, as shown next. Get the files by clicking “Download ZIP” at the top right.



The browser will tell you when the download has finished. Open up a new Terminal and issue the following command to unpack the files, and then delete the zip package to clear space.

```
unzip Downloads/makeyourownneuralnetwork-master.zip
rm -f Downloads/makeyourownneuralnetwork-master.zip
```

The files will be unpacked into a directory called makeyourownneuralnetwork-master. Feel free to rename it to a shorter name if you like, but it isn't necessary.

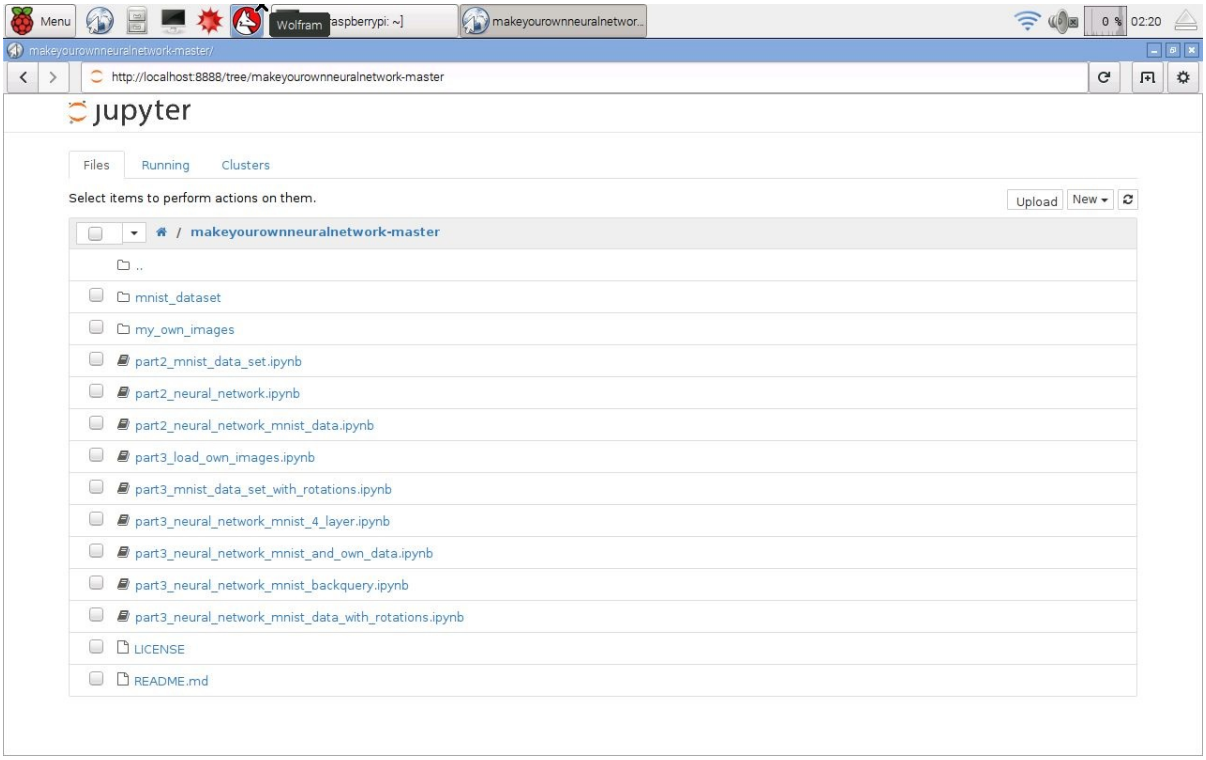
The github site only contains the smaller versions of the MNIST data, because the site won't allow very large files to be hosted there. To get the full set, issue the following commands in that same terminal to navigate to the mnist_dataset directory and then get the full training and test datasets in CSV format.

```
cd makeyourownneuralnetwork-master/mnist_dataset
wget -c http://pjreddie.com/media/files/mnist_train.csv
wget -c http://pjreddie.com/media/files/mnist_test.csv
```

The downloading may take some time depending on your internet connection, and the specific model of your Raspberry Pi.

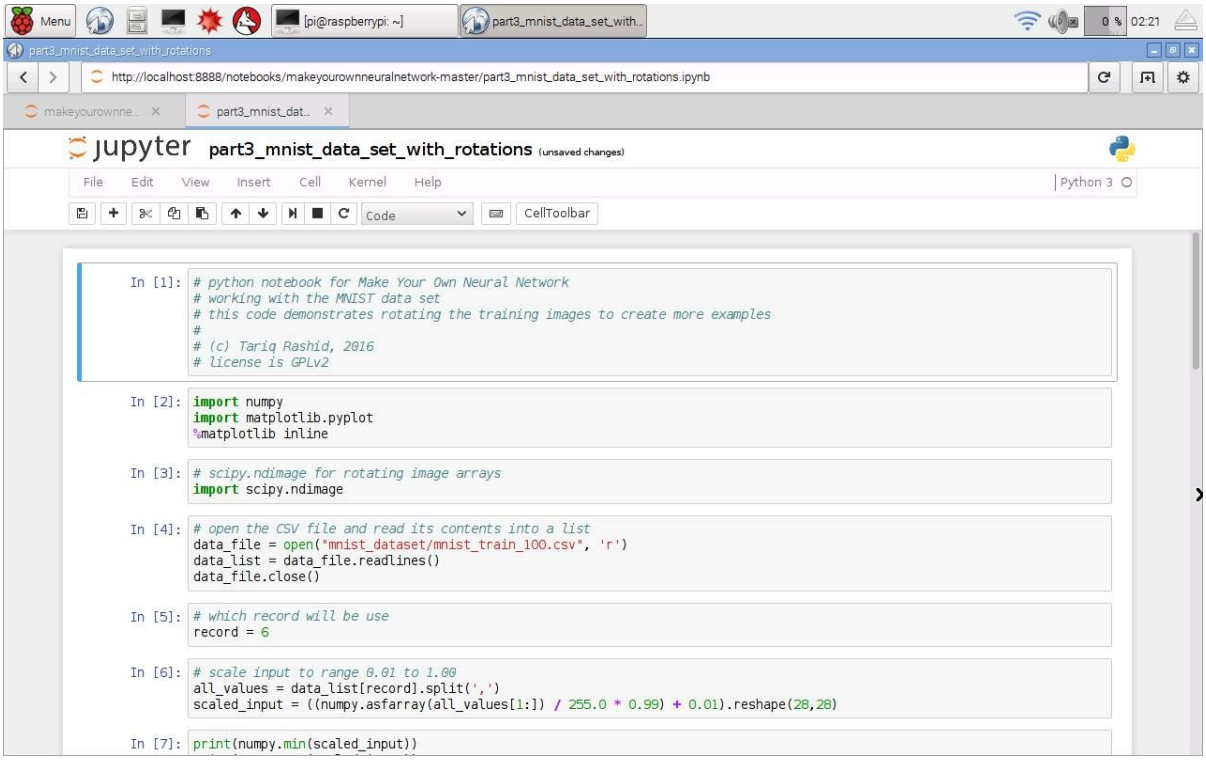
You've now got all the IPython notebooks and MNIST data you need. Close the terminal, but not the other one that launched IPython.

Go back to the web browser with the IPython starting page, and you'll now see the new folder makeyourownneuralnetwork-master showing on the list. Click on it to go inside. You should be able to open any of the notebooks just as you would on any other computer. The following shows the notebooks in that folder.

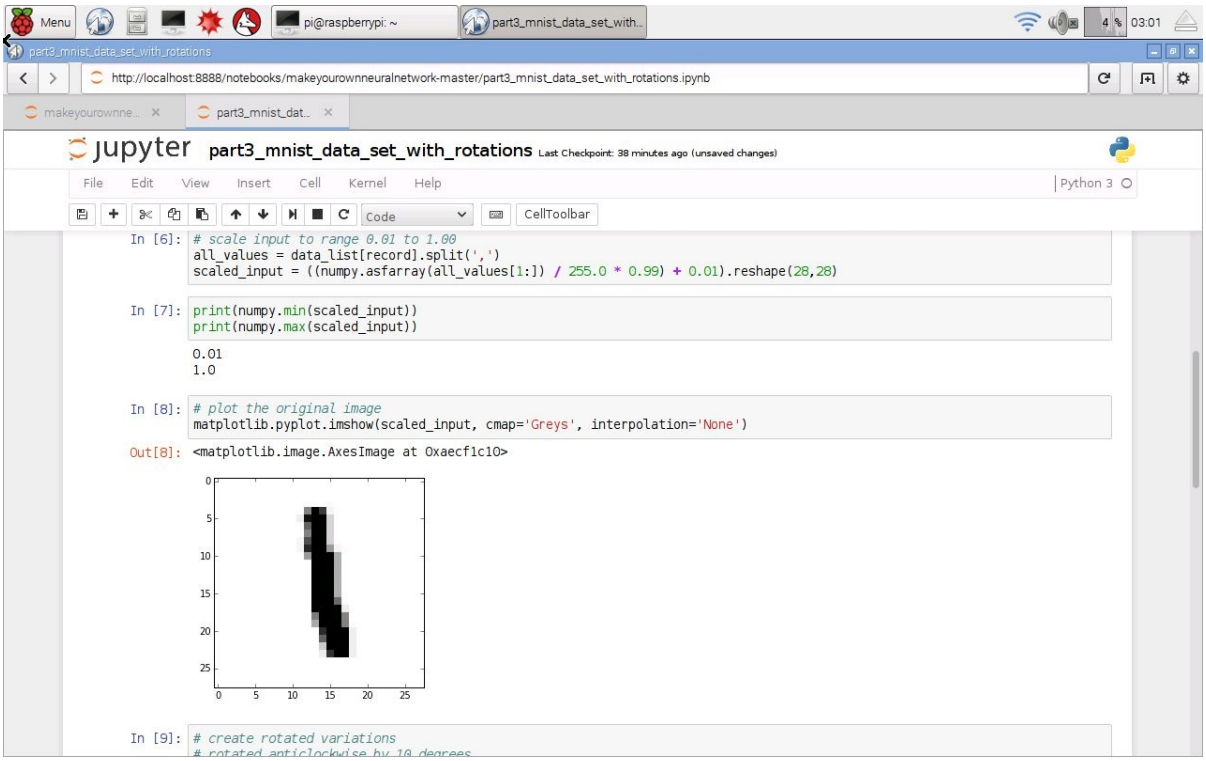


Making Sure Things Work

Before we train and test a neural network, let's first check that the various bits, like reading files and displaying images, are working. Let's open the notebook called "part3_mnist_data_set_with_rotations.ipynb" which does these tasks. You should see the notebook open and ready to run as follows.



From the “Cell” menu select “Run All” to run all the instructions in the notebook. After a while, and it will take longer than a modern laptop, you should get some images of rotated numbers.



That shows several things worked, including loading the data from a file, importing the Python extension modules for working with arrays and images, and plotting graphics.

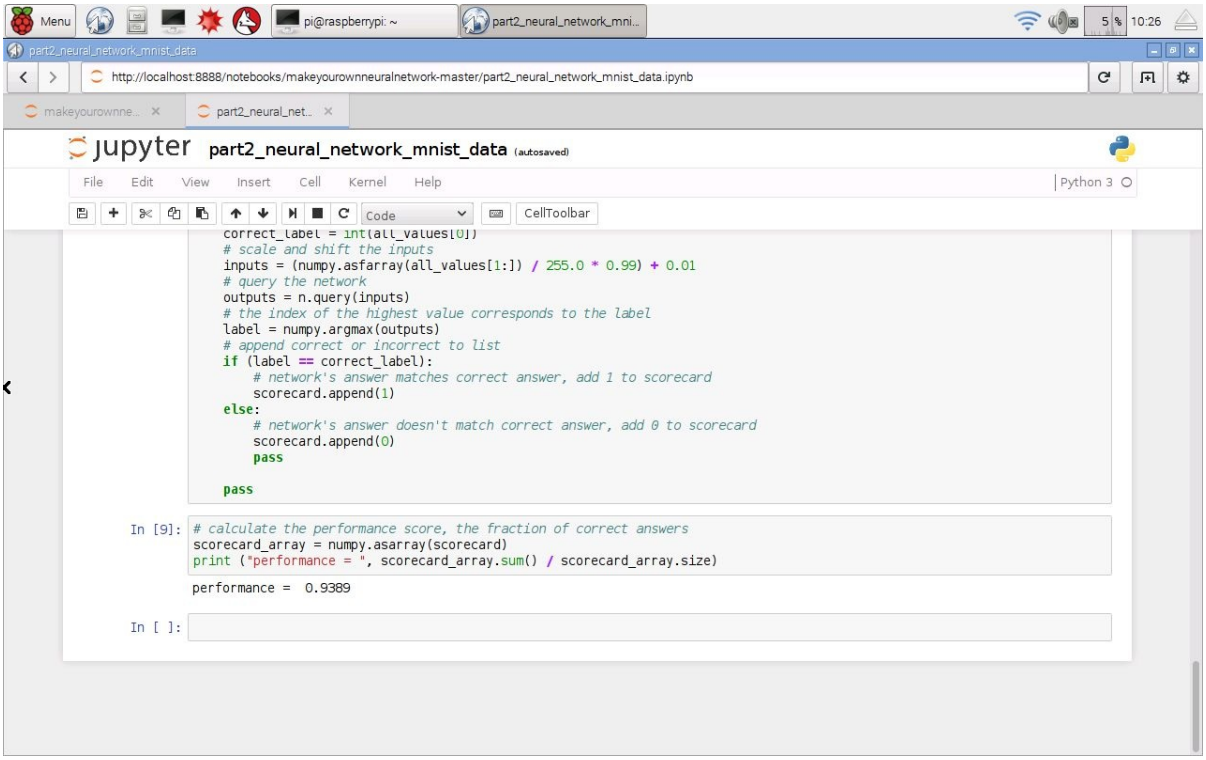
Let’s now “Close and Halt” that notebook from the File menu. You should close notebooks this way, rather than simply closing the browser tab.

Training And Testing A Neural Network

Now lets try training a neural network. Open the notebook called “part2_neural_network_mnist_data”. That’s the version of our program that is fairly basic and doesn’t do anything fancy like rotating images. Because our Raspberry Pi is much slower than a typical laptop, we’ll turn down some of parameters to reduce the amount of calculations needed, so that we can be sure the code works without wasting hours and finding that it doesn’t.

I’ve reduced the number of hidden nodes to 10, and the number of epochs to 1. I’ve still used the full MNIST training and test datasets, not the smaller subsets we created earlier. Set it running with “Run All” from the “Cell” menu. And then we wait ...

Normally this would take about one minute on my laptop, but this completed in about **25 minutes**. That's not too slow at all, considering this Raspberry Pi Zero costs 400 times less than my laptop. I was expecting it to take all night.



Raspberry Pi Success!

We've just proven that even with a £4 or $5 Raspberry Pi Zero, you can still work fully with IPython notebooks and create code to train and test neural networks - it just runs a little slower!