NIHR | Applied Research Collaboration South West Peninsula

@penCHORD_UoE
@peninsula_ARC

Module 7 : Machine Learning
Session 7D : Introduction to Neural Networks
Dr Daniel Chalk

"Deep Joy"

HSMA 5

#hsma5isalive

Neural Networks

Neural Networks (or more specifically for our purpose, *Artificial Neural Networks*) are networks of *neurons* (nodes) with *connections* between them, and *signals* being sent along these connections.

Neural Networks mimic the way in which the human brain works, and were developed originally to try to better understand how the brain works.

The irony is that whilst Neural Networks have been proven to be *superb* structures for artificially intelligent learning, we don't understand *why* they work… (although there is a growing field of "Explainable AI")

So let's just pretend it's magic :)

GPUs

If you're going to dive deep into the world of Neural Networks, you may want (or eventually need) to consider using a *GPU*.

A GPU is a *Graphical Processing Unit*. They're designed primarily for gaming (and for high-end video and graphics editing).

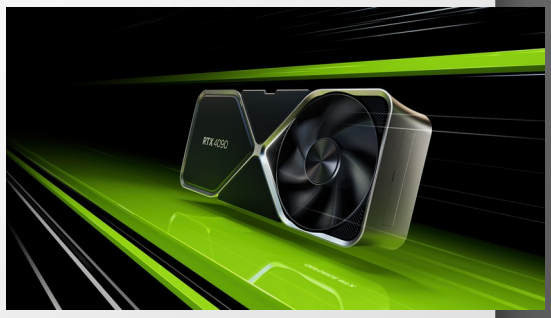
GPUs are excellent at taking large amounts of inputs and processing them all simultaneously very quickly. As you'll see as we explain Neural Networks, this turns out to be good for the kind of processing we need.

However, they're not always the best choice – whilst they process quickly, it takes a lot of time to get the data onto (and back off) the GPU, so it really makes most sense for big and complex neural networks.

GPUs

If you want to get a GPU to use for Neural Networks, you need to ensure :
- It's a NVidia GPU
- It's CUDA-Enabled (CUDA = Compute Unified Device Architecture) – these are the GPU's equivalent of CPU cores that do all the parallel processing. But not all GPUs (even NVidia GPUs) have them. You can find a list here, along with the "compute capability" score :
https://developer.nvidia.com/cuda-gpus
- You have relatively deep pockets…

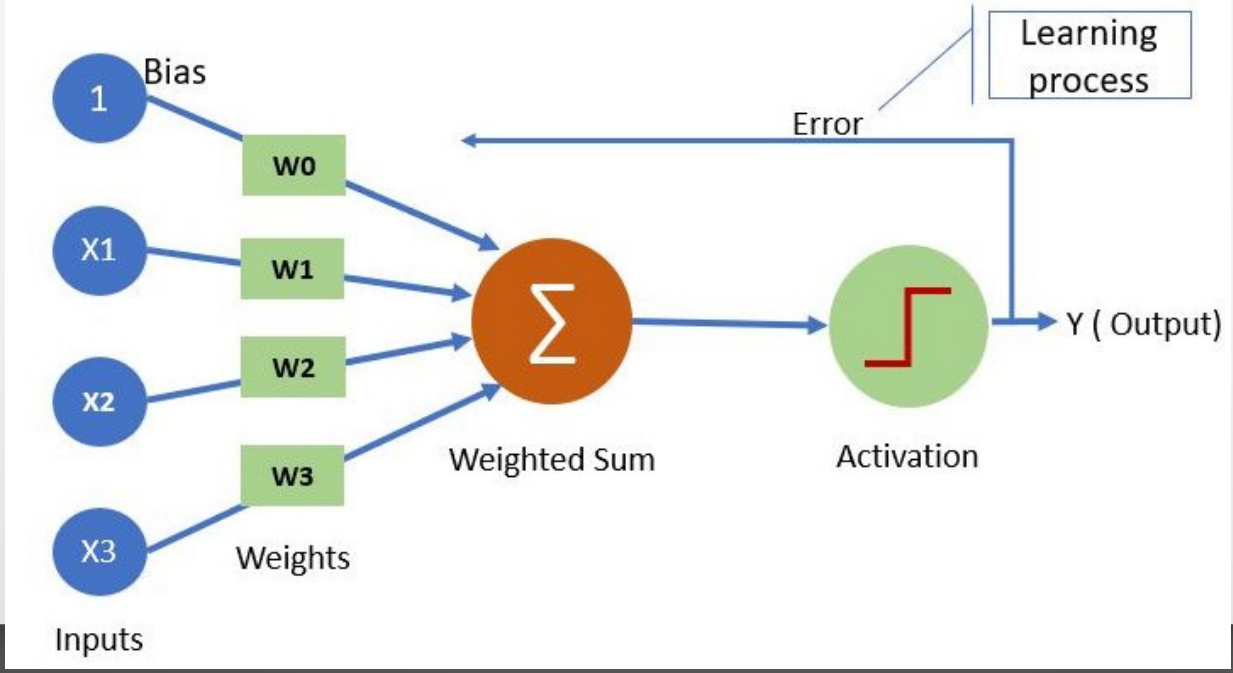
There are alternatives if you want to use a GPU for Neural Networks but can't practically buy a CUDA-enabled one. Online services such as Google CoLab allow you to use a GPU in the "cloud".



The Neuron

Neural Networks are made up of Neurons (originally named *Perceptrons*). Each Neuron performs a very simple task – to take all of its inputs (each multiplied by a *weight* representing the strength of a connection), add them all up, and spit them out according to an *Activation Function.*

The Activation Function takes the raw weighted sum, and converts it in some way.

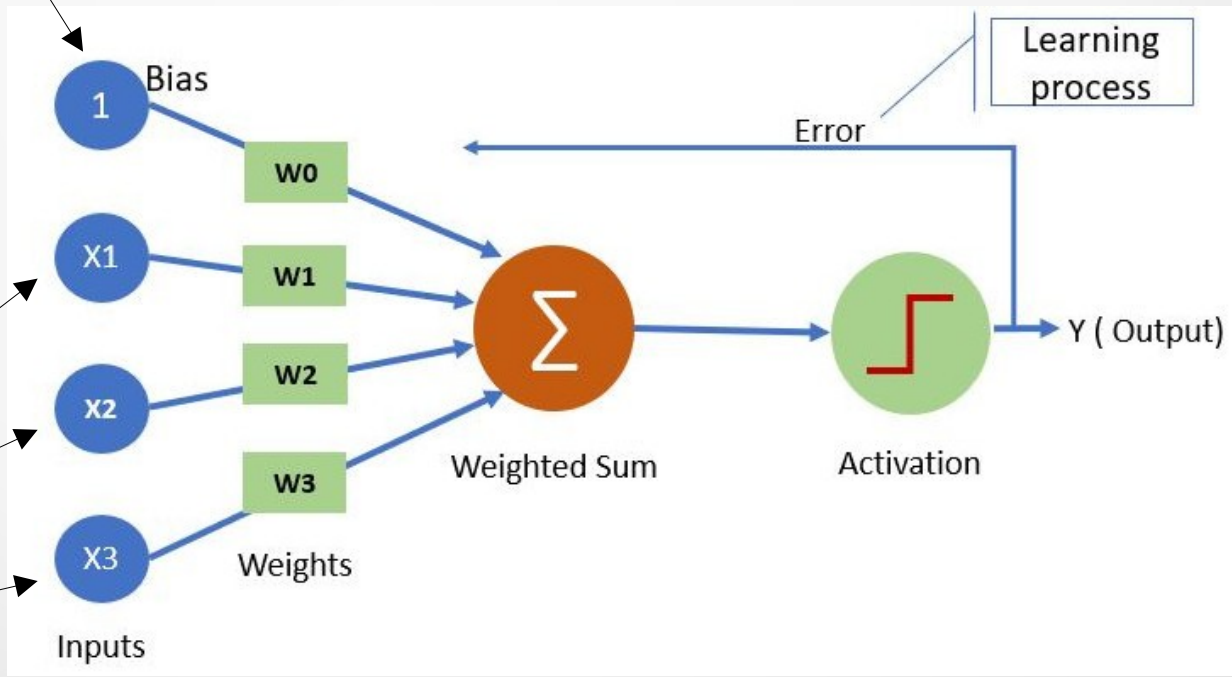


The Neuron

The Perceptron was first conceived by American Psychologist Frank Rosenblatt in 1957. He was trying to simulate the workings of the human brain in order to create an early form of machine learning, in which a computer learned by trial and error.

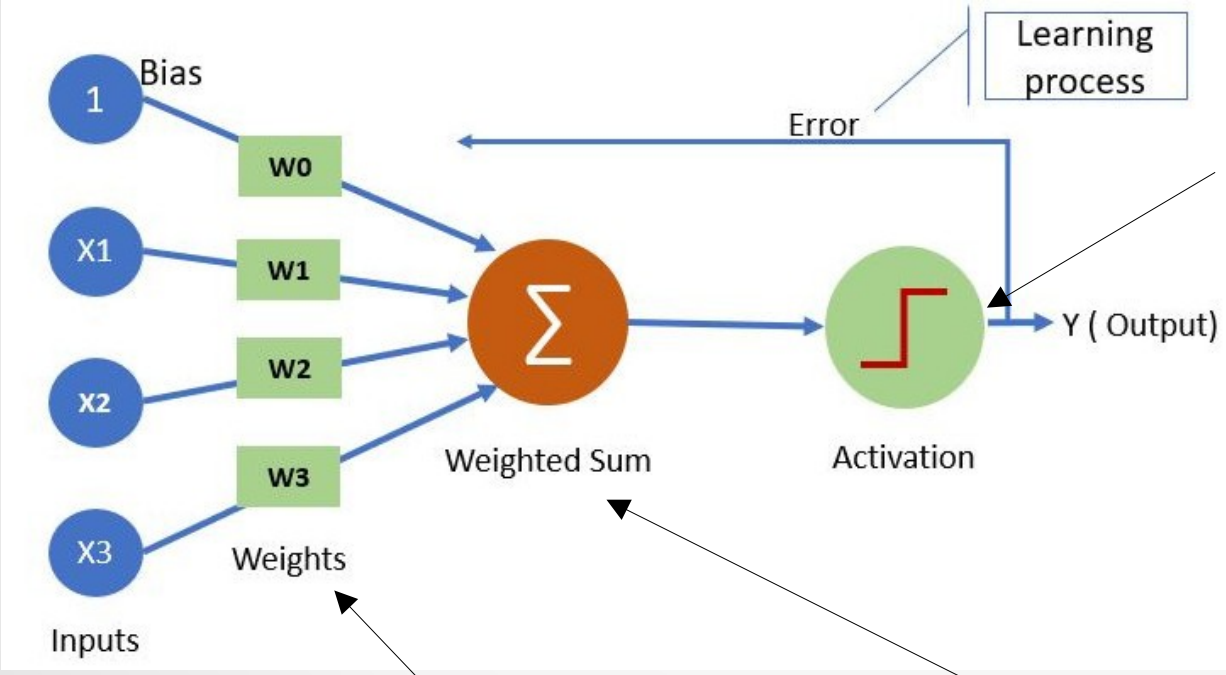
The Neuron

We have one input as a “Bias”. This is a bit like a constant in a linear function. It allows us to “shift” the activation function, which may help us learn better (e.g. if we shifted our activation function graph over to the left a bit, or the right a bit, it might be a better fit. The network will do this automatically – by including it as an input with a weight, the network can play around with it automatically to improve learning performance.



These represent values from our *features* (data in our “columns” – the information from which we’re trying to make a prediction)

The Neuron



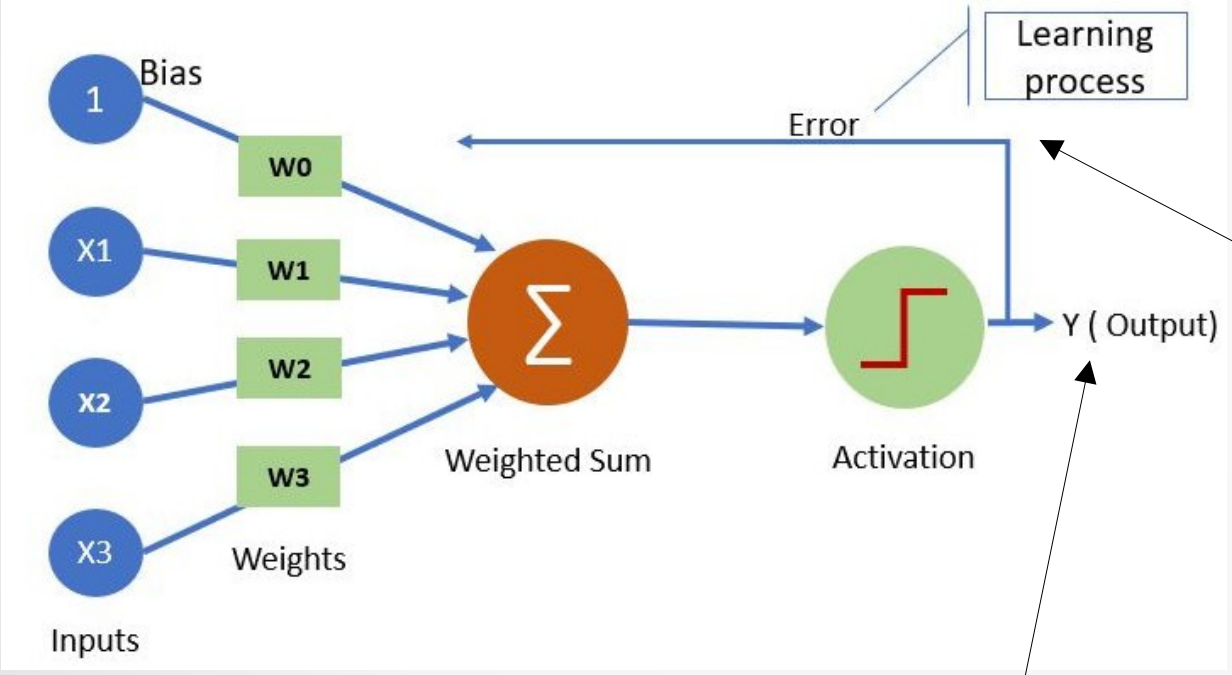
Once the neuron has performed its weighted sum of inputs, it fires it out via an “Activation Function”. This function transforms the output in some way (we’ll talk about that more shortly)

The original activation function was a “Step Function”, in which the output was either 0 or 1, depending on whether or not the weighted sum was higher or lower than a threshold value. This mimics a neuron in the brain being active or inactive.

Inputs are connected to a neuron along with a “weight”. These weights represent the relative importance of that input to the final calculation made by the neuron. The network plays around with these weights whilst it’s learning. Weights start with completely *random* values.

The neuron performs a “weighted sum” – it takes each input multiplied by its weight, and then adds up all those numbers

The Neuron

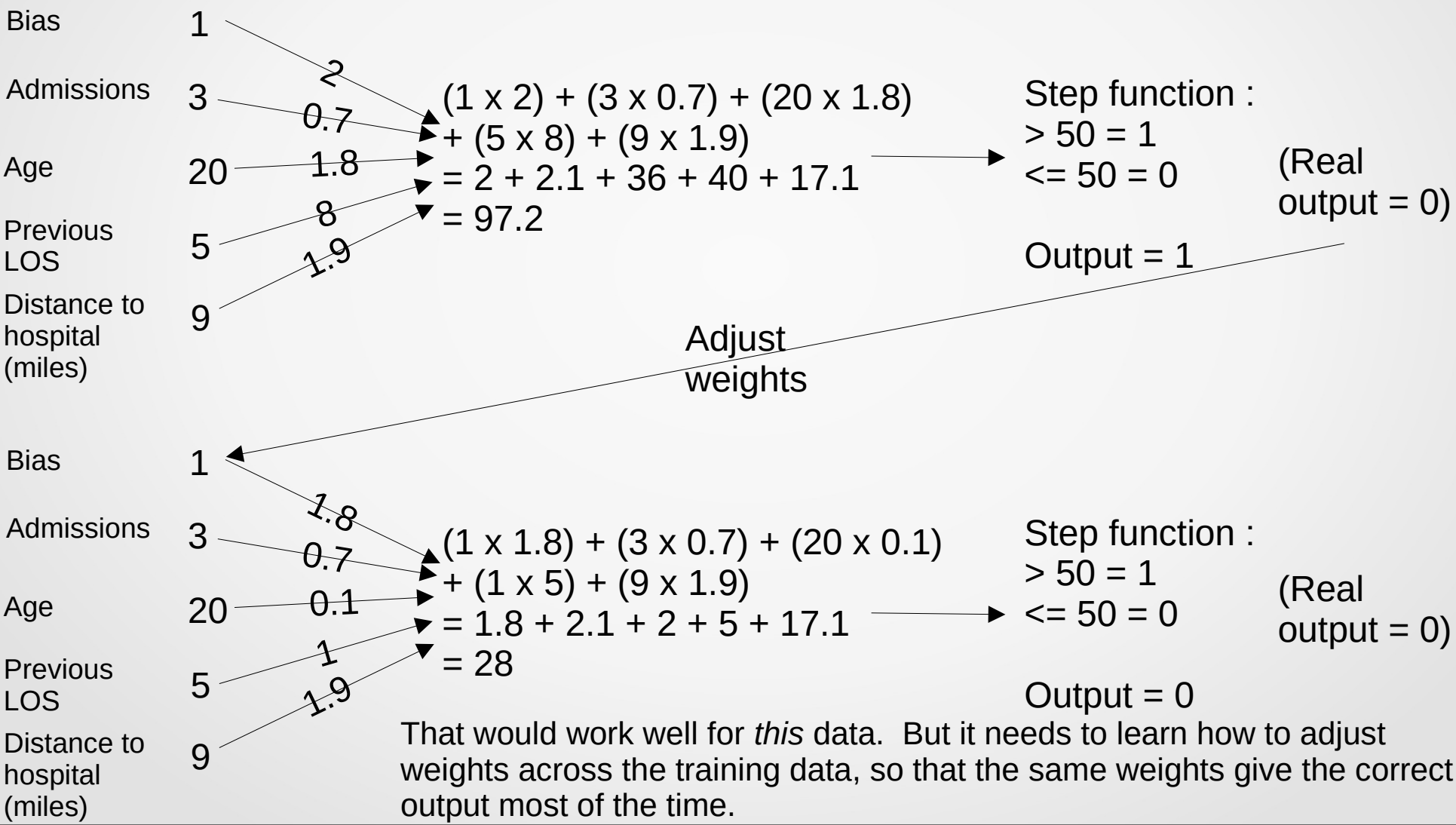


Key to learning in a Neural Network is “Error” – how much was the output (prediction) right or wrong? The “learning” of a Neural Network takes place as the network adjusts the weights to try to get the output prediction closer to the true value (in other words, minimise the error).

The Activation Function spits out an output (which is the weighted sum having been fed through the Activation Function). This output represents a “prediction”. This might be a classification prediction (is it this thing or this thing? - 0 or 1) or a probability (e.g. it’s 0.7 likely it’s this thing)

The Neuron - Example

Let's look at a simple example
to see how this works.



That would work well for *this* data. But it needs to learn how to adjust weights across the training data, so that the same weights give the correct output most of the time.

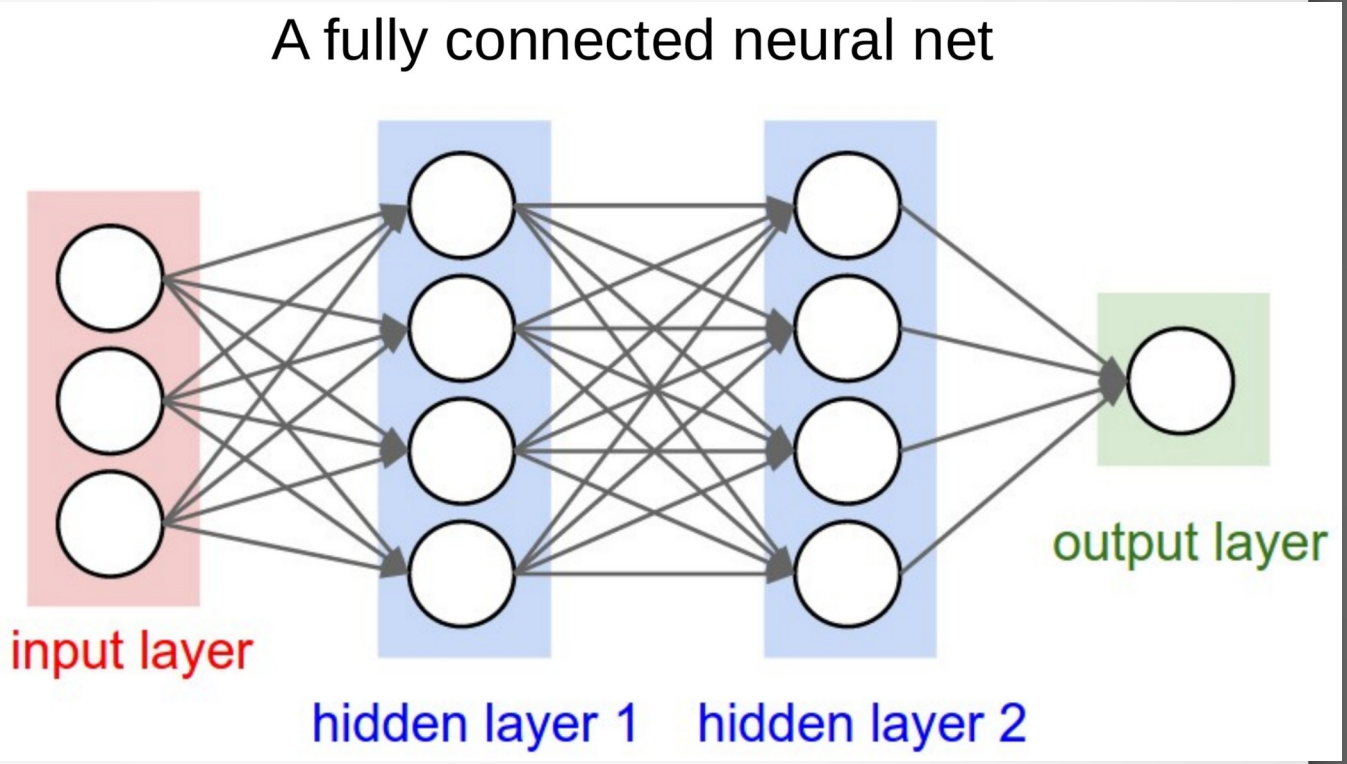
A Neural Network

A *Neural Network* is simply a network of *neurons* that are connected together across different *layers*. The outputs from neurons in one layer feed into neurons in the next layer (acting as their inputs).

A *fully connected* neural network is one in which every neuron in one layer is connected to every neuron in the next layer.

Each connection in the network has its own weight.

The network is trying to learn how to adjust the weights *across* the network so that it predicts the correct output for a given set of inputs, most of the time.



The multiple layers of a Neural Network give the name to the sub-field of AI in which they sit : **Deep Learning**

A Neural Network

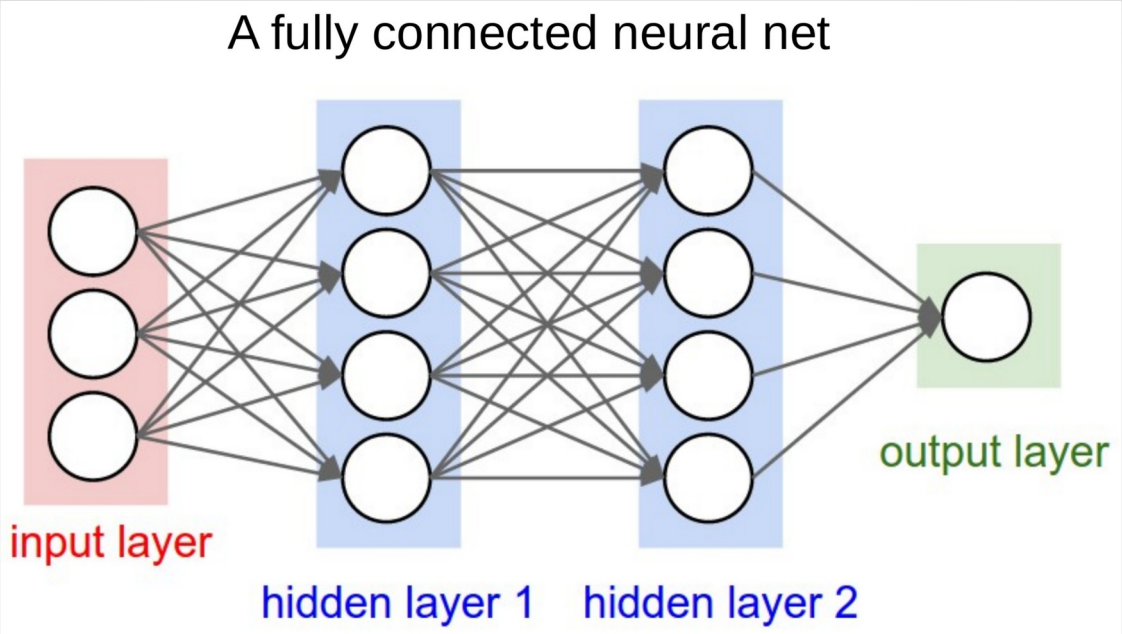
Whilst the structure of Neural Networks is relatively simple, the reason why they so consistently work so well for so many problems is not well understood. The "magic" of the network changing its weight values such that a set of numbers are transformed and manipulated until eventually coming together to form an output value is **black box**.

(The irony is that Neural Networks were conceived as a way of trying to better understand the human brain. It turns out that they, like the human brain, are extremely powerful. But also, like the human brain, we don't really understand why.)

The black box nature of Neural Networks can be problematic, as it means that you cannot explain precisely *why* your Neural Network performs so well for a problem. This can affect stakeholder trust in the model (although their increasingly wide usage helps).

There is an emerging field of "Explainable AI", particularly using things called *Shapley Values* which try to unpick the contributions of various features. We don't cover them in Phase 1, but intend to run a Masterclass covering this in Phase 2.

A Neural Network



An *Input Layer* is the first layer of the network, and takes the (standardised or normalised, likely normalised for a Neural Network) values of the features in your data.

An *Output Layer* is the final layer of the network, and, in a classification network, represents the prediction as to whether an example belongs to a class (where the output value would represent a probability, and a *threshold* (default of 0.5) is used to determine whether it belongs to class 0 or 1. This is typically a single neuron, but can be more than one (e.g. if you want to make an assessment of how confident it is that an example belongs to one of three classes, such as three types of flower given an image). For regression problems (where you're predicting a continuous value), the output is typically left raw.

Hidden Layers are the layers in the middle where most of the magic happens.

Activation

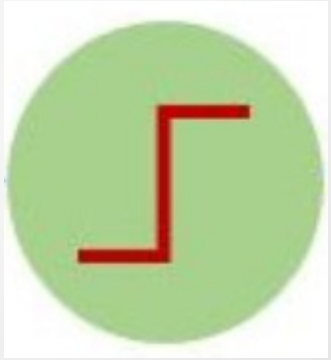
There are lots of different activation functions. You *can* have different activation functions for every neuron, but typically you'll have the same one for most of the network, and maybe a different one to deal with your output.

Let's have a look at some commonly encountered activation functions.

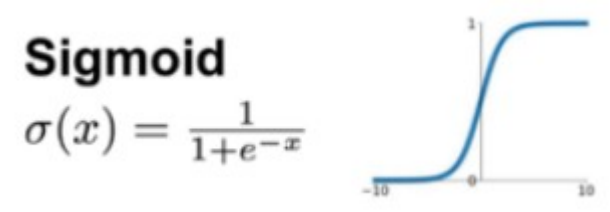
Activation

Linear – this activation function essentially does nothing (it just passes the weighted sum through as-is). This can be useful for the final neuron (output layer) of regression problems where you're trying to predict a continuous variable (eg house prices).

Step – if value is above a threshold, output 1, otherwise output 0. Never used now, and is nothing more than a historical artefact (being the very first activation function).



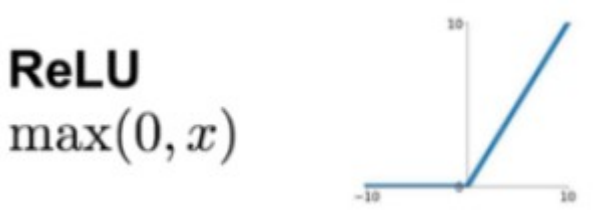
Activation



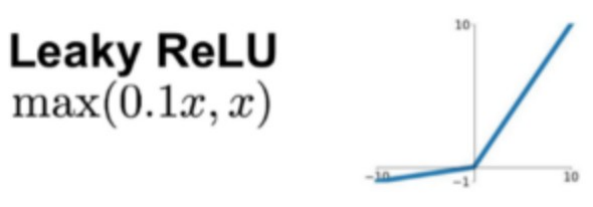
Sigmoid – scale output to between 0 and 1 using logistic function (as we saw in last session). Good for producing probabilities, and therefore **often used as final output neuron in classification networks**.

tanh – similar to Sigmoid but scales between -1 and 1. Common in older networks, but rarely used now.

Activation



ReLU (Rectifying Linear Unit) – keep positive values unchanged, and convert all negative values to 0. Allows for modelling of non-linear functions and is widely used (it turns out this very simple way to introduce non-linearity is sufficient for Neural Networks to learn very complex patterns). **Probably the activation function you will use the most.**



Leaky ReLU – as with ReLU, but rather than converting all negative values to 0, it simply reduces their scale (so you still have negative values, but the range becomes compressed). Can sometimes be useful if the weights your network are frequently being set to negative values (which can happen with high learning rates), as ReLU will set them to 0 (thereby "switching them off"). This can lead to an effect where the network increasingly shuts down and can even reach a state where all neurons are dead, turning the network into a constant function – this is known as the *Dying ReLU Problem*.

See https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24

Activation

In general, for most Neural Networks you would build, you should likely :

1. start with ReLU for most of the network, and use Sigmoid for the output layer

2. consider switching ReLU for Leaky ReLU if you encounter the Dying ReLU problem

3. consider switching Sigmoid to something called *SoftMax* if you have multiple outputs and want them to add up to 1 (e.g. you want the total of your probabilities for "this is dog", "this is a cat", "this is a fish" etc to add up to 1).

Loss and Backpropogation

The magic of a Neural Network lies in how they can learn by adapting the weights of their connections over time.

They do this using a process known as *backpropogation.* Backpropogation distributes the error (known as the *loss* – how far off the NN was in terms of the output it predicted vs the true output) across the network. In other words, it works out which connections / weights were most to blame for the error, and adjusts them accordingly.

So we need a way to calculate the loss (loss function) and a way to distribute the error (backpropogation).

There are various ways of measuring loss and also of optimising the backpropogation. Let's look at some of them now.

Loss Functions

Loss functions are used to calculate the *loss* (error – how far off the network's prediction was).

The two most commonly used Loss functions are :

- *Mean Squared Error Loss*
- *Cross Entropy Loss*

Mean Squared Error Loss

Mean Squared Error (MSE) Loss is commonly used for regression problems (where you're trying to predict a continuous value, such as house prices, or length of stay in a hospital).

The function works simply by calculating the difference between the observed (real) and predicted value, and squaring it.

e.g. Real length of stay = 20 days
Predicted length of stay = 16 days
MSE = $(20 – 16)^2 = 4^2 = 16$

We square the error to allow for the loss (error) being either side of the real (observed) value (e.g. - predicting 4 days over or 4 days under would be treated as the same scale of loss (error)).

Cross Entropy Loss

Cross Entropy Loss is commonly used for classification networks (predicting whether an example belongs to one class or another).

Binary Cross Entropy Loss is used when the output is *binary* (two possibilities – e.g. “survived” or “died”). It’s likely many of your own networks would fall into this category.

The calculation of Cross Entropy Loss is complicated, and you don’t need to understand the details, but broadly it assigns rapidly increasing penalties the further the predicted probability differs from the true probability. This means that a prediction that is confident but incorrect will suffer a huge penalty (eg output of 0.9 (confident class 1) when true class = 0), compared to a prediction that classified incorrectly but was less confident (eg output of 0.55 (marginal prediction of class 1) when true class = 0).

Backpropogation

Backpropogation describes the process by loss is distributed back through the network, to work out how much “blame” should be assigned to various parts of the network.

Backpropogation was conceived by cognitive psychologist and computer scientist Geoffrey Hinton in 1986. This revolutionised their use, as it provided a means of being able to train Neural Networks well.

The mathematics behind it are complicated (unsurprising, given it took nearly 30 years for someone to come up with it!). But the good news is you don’t need to understand it to build Neural Networks.

Think of it as a “blame assignment” algorithm.

Backpropogation uses an *Optimiser* to update weights in the network. Let’s talk about optimisers now.

Optimisers

An *optimiser* decides how to adjust each weight (magnitude and direction of change) in the network based on the loss that has been backpropogated. The aim is to adjust the weights such that the loss is reduced next time.

The optimiser is the “correction” algorithm to try to improve the weights that got blamed for contributing to the loss.

As a builder of Neural Networks, you need to decide which optimiser you will use.

There are many different optimisers available, but we will cover here three of the main ones that are most commonly used.

Stochastic Gradient Descent

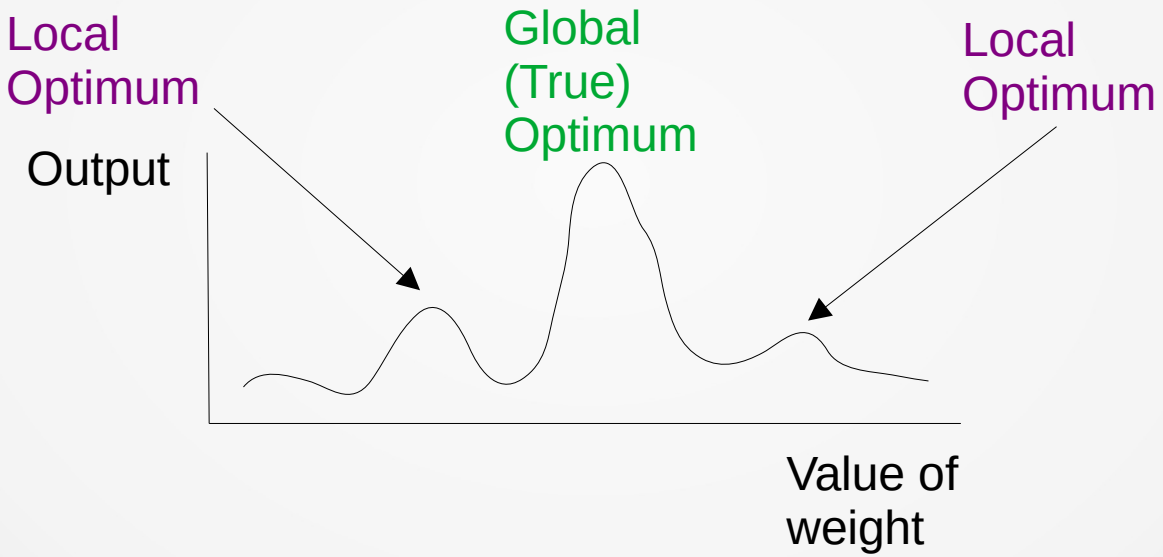
Stochastic Gradient Descent is a classic optimiser in which it adjusts weights in response to how much the changes alter the output of the network (as well as identifying the direction of change that improves things).

It makes *bigger* changes the further away it believes it is from an optimum value (it determines it is further away if it finds that the output is highly sensitive to small changes to the weight).

It makes *smaller* changes the closer it gets to an optimum value (when small changes to the weight don’t result in big changes to the output).

Stochastic Gradient Descent

A problem with Stochastic Gradient Descent is that, by making these “big changes” when far away, and “small changes” when getting close, it can get trapped in a local optimum rather than the true optimum.



We can use something called *momentum* to get the algorithm to explore more, to try to prevent it getting stuck in one of the local optima. Think of it as giving it “a bit of a shove”.

RMSProp and Adam

RMSProp (Root Mean Squared Propogation) is a ‘classic’ optimiser for Neural Networks. It works similarly to Gradient Descent, but has a *dynamic learning rate* in which the which the step size (the amount a weight changes) varies for different parameters at different times, depending on the rate of change being observed.

Adam is the most commonly used optimiser used for Neural Networks today. In fact, “if in doubt, use Adam” is a good mantra. Adam combines the benefits of RMSProp with another extension of Gradient Descent (AdaGrad – Adaptive Gradient Algorithm) to have a more sophisticated way of adapting learning rates.

We’re in danger of getting into deep maths here; all you really need to know is :
- Adam is usually the best option these days
- You could try RMSProp to see how performance compares
- Stochastic Gradient Descent isn’t really used now, but the core concepts have passed down to Adam and RMSProp

A picture of some fluffy kittens

Iterations and Epochs

When training a Neural Network, we have multiple *iterations,* and in each iteration we pass through a *batch* of our data (it's rare we pass everything through all at once – see next slide). Once all iterations have been completed (we have passed through all of our batches of data), we have completed an *epoch*.

In each iteration, we :

1. Pass our input data (X) through to try to predict the output (y) – this is known as a *Forward Pass*
2. Calculate the loss (error) between predicted and observed (real) values of y
3. Backpropogate the loss and use the optimiser to update the weights

The good news is that *Keras* (the API with which we interact with *TensorFlow* – the Neural Network framework we'll be using) does all of the above for us – we just specify the structure of the network and the algorithms (loss function, optimiser etc) that we want to use. The common alternative *PyTorch* requires you to run steps 2 and 3 separately and manually.

Depending on the problem, and the data you have, you may need a lot of epochs. 10 is a good number to start with, but it may be that you need hundreds or even thousands to learn effectively. But we need to be careful of *overfitting* (where the network has learned things that are too specific to the training data, and has lost generalisability) and *underfitting* (where we haven't learned enough)

Batches and Batch Size

We don’t typically pass all of our data through the network at once. This is because, although passing more data through at a time means we train quicker, performance tends to worsen, because the model is getting too much information at once, and it’s not properly able to learn the “nuances” within.

Therefore, we typically split our data up into *batches*. Essentially, we chop our data up, and shunt through a batch of it at a time.

The size of the batch is up to the modeller (and something you want to play around with for your model). However, there is an opinion (popularised by Yann LeCun – a French Computer Scientist and a big name in the field of Deep Learning) that you shouldn’t use batch sizes of greater than 32.

But try things out yourself! Neural Network building is as much (if not more) an art form than a science. And if you’re using a GPU, it of course thrives with big batch sizes (computationally – that doesn’t mean you should do it!)

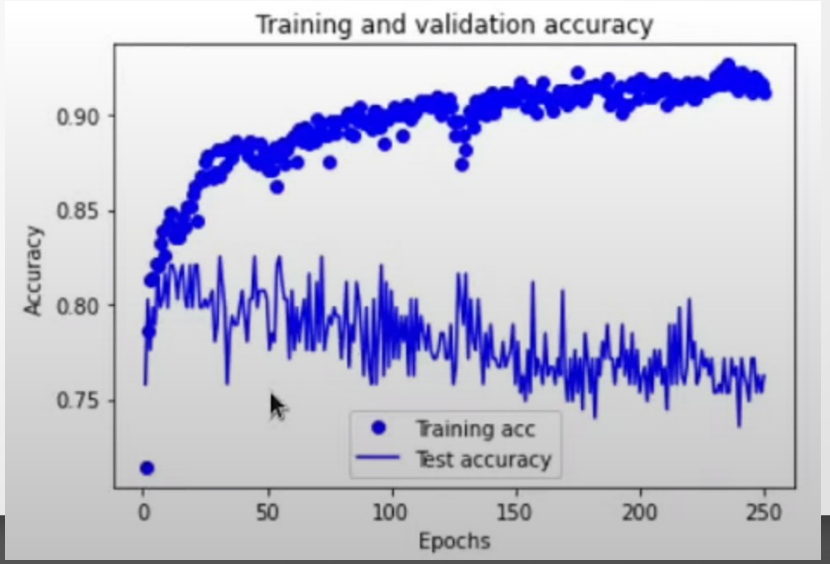
“Training with large minibatches is bad for your health.
More importantly, it's bad for your test error.
Friends dont let friends use minibatches larger than 32.”

Overfitting

If we let a Neural Network train for long enough (enough iterations), we will get excellent (possibly even 100%) accuracy on the training data.

But usually this means we have overfitted, and the network will not perform well on other data. We can usually tell, because we will have a very good performance on the *training set*, and a much worse performance on the *test set*. We'll also typically see a test set accuracy that initially goes up before gradually reducing (whilst the training accuracy gradually increases).

We can take action to try to prevent overfitting. **However**, typically we want to initially start with a Neural Network that **IS** overfitting to ensure we're learning **enough** to avoid underfitting. Then, we can look at gradually reducing the training to avoid overfitting.



Prevent Overfitting - Regularisation

Once we've got a model that we're confident is learning sufficiently to avoid underfitting (because we've got high training set accuracy and it's overfitting), then we can try to tackle the overfitting problem.

We can do this using *Regularisation* – a set of approaches that include :
1. Reducing the complexity of the model
2. Reducing the amount of time we allow the model to train (reduce number of iterations)
3. Adding Dropout

There is also something called L1 and L2 Regularisation, but this doesn't tend to be widely used now.

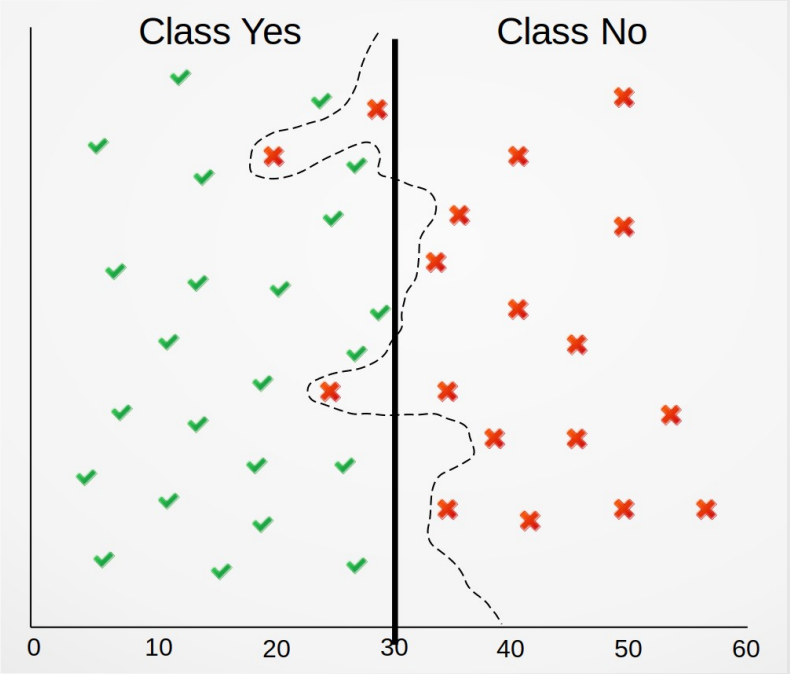
Let's talk about each of the three main approaches in turn.

Reducing complexity

Another way to think about a classification model that is overfitted is one that has come up with a function (“dividing line”) that is too complicated (and unique to the training data).

So one way in which we can prevent overfitting is by *simplifying* the model, thereby reducing the complexity of the function (“dividing line”) that it learns.

This might include reducing the number of hidden layers, reducing the size of each layer etc.

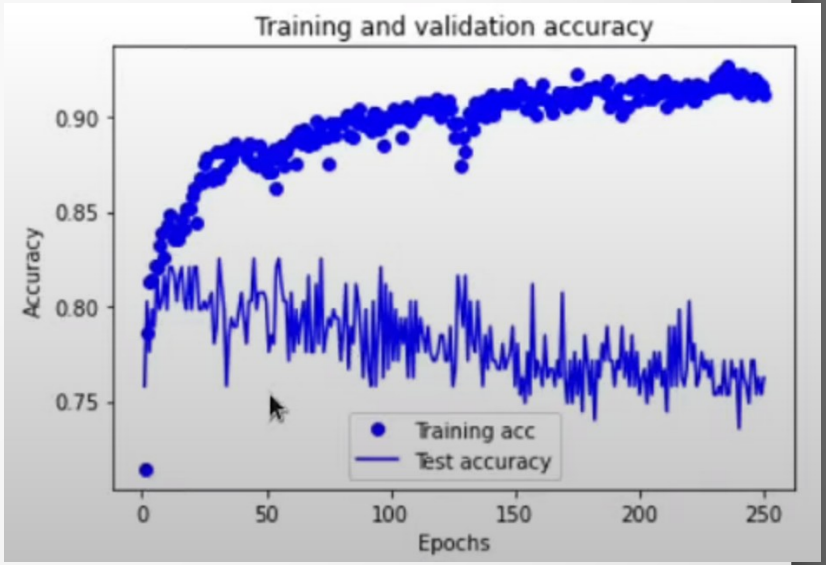


Reducing training time

Another tactic we could use is to stop the training before it starts overfitting.

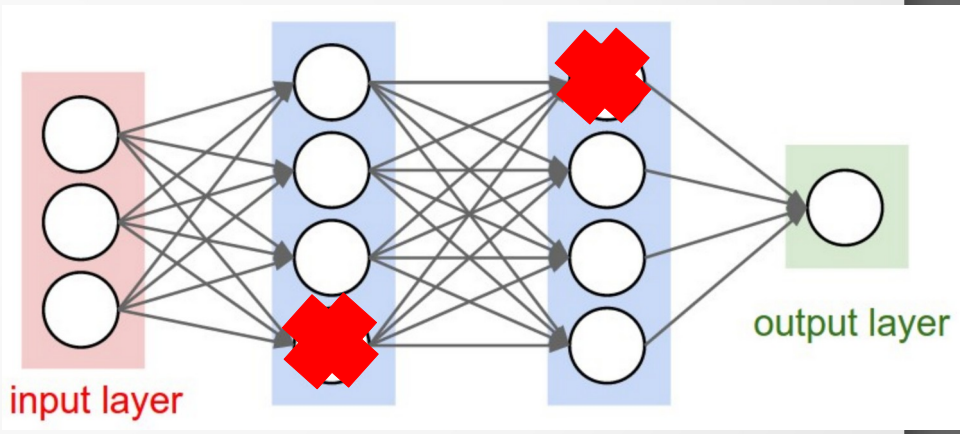
We can either do this manually (by looking at when our test accuracy tends to start decreasing), or using a callback known as *EarlyStopping*.

EarlyStopping allows us to specify the minimum amount of change we want to see in a performance metric between epochs, and how long we're prepared to wait for things to improve sufficiently.



Dropout

Dropout is another common strategy for avoiding overfitting. In each epoch (and only during training), Dropout randomly switches off a percentage of neurons in your network (commonly between 20% and 50% - you can specify this). The neurons it chooses are different in each epoch. (Strictly speaking, it switches off the weights by adding a dropout layer and changing the weights coming out of the neurons to 0).



This means that, during training, the network doesn't become over-reliant on certain weights in your network (which is what happens when a model is overfitted).

When the fitted (trained) model is deployed, there is no dropout.

Tensors

Whilst we're using simple numbers here to describe how a neural network works, in reality the data being passed through a neural network is a bit more complex.

The numbers that we feed into a neural network need to be shaped as *Tensors*. A Tensor is quite a complex concept to understand, but a good way to think of a tensor is as a generalisation / abstraction of scalars and vectors (which are themselves Tensors).

A scalar represents a quantity that has magnitude but no direction. A vector has both scale and direction. Both of these are Tensors, but Tensors can also go into multi-dimensional space.

But don't worry about the complexities of this. All you need to know is that a Tensor is the shape that your numerical data needs to be in order to be pushed through a Neural Network. The packages you will use have methods to get your data into the right shape.

TensorFlow and Keras

Building and using Neural Networks were, until fairly recent times, the domain of expert mathematicians and Computer Scientists.

But in recent years, excellent packages have emerged that allow many more people to be able to (relatively) easily build their own Neural Networks, by hiding much of the “pure mathematical” complexity.

Two of the most well-established packages are *PyTorch* and *TensorFlow (developed by Google)*. In this session, you will have practice using TensorFlow using the *Keras* API, which sits on top of TensorFlow and makes things a bit more user-friendly.

Exercise 1

In your groups, you will now (after a 10 minute break) :

1. Work through the notebook hsma_neural_nets.ipynb. You should work through this as a group, with one person sharing their screen, and make sure that you all understand how everything is working.

2. Once you're happy that you understand all of this, you should (in your groups) build a Neural Network that attempts to predict whether a patient receives clot busting treatment for stroke, using the same data that you used when attempting this with a Logistic Regression model in the previous session (using the notebook HSMA_exercise.ipynb – take a copy of it using "Save Notebook As…" - you just want the first cell that downloads the data).

You have 90 minutes (+ your 10 minute break now). How well can you get your model to predict? Can you improve your accuracy by avoiding or reducing over-fitting? We'll ask a few groups to share what they did when we come back.