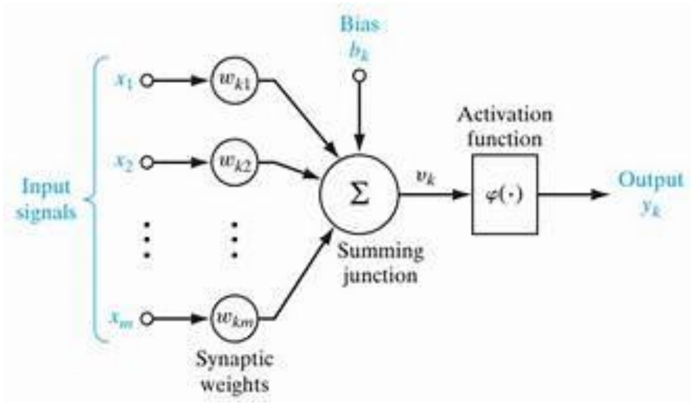
Neural Networks (based in Python 3.8.0)

Neural networks are essentially an iterative approximation for a given set of inputs and an output. Through each iteration we adjust the values which affect what choices it makes relative to the inputs relating to the outputs. These are called synaptic weights; the synapses are essentially connections between neurons/nodes. In this first example, we'll take a look at a simple single perceptron model. There is some math involved, but it's nothing too complex yet, just basic high school calculus. But this will change with more complex models later on, you'll need to have a good grasp on university level linear algebra and calculus. You have been warned!

So, let's crack on with the single perceptron. The model that will be implemented is as follows:



This might look complicated but it's really not, machine learning and AI topics are clouded with a lot of jargon so I'll do my best to break it down for you as the actual concepts, rather than throwing random words at you.

First off, we have the input signals, these are essentially values that represent the data that is being analysed by the network. Generally, these are vectors or matrices which have a format specific to the type of network being used. A nice analogy to this is thinking about a network that analyses images, the image you want to classify could be represented by a matrix of values, each entry in the matrix being a colour or chromatic value. You could even represent this with a single vector if you have a regular way to split the sections/lines that make up the image. Each of the inputs is a value of a specific entry in that matrix or vector, something that is measurable that can be analysed and compared.

Next, the synapses and synaptic weights. The synapses are the connections between the inputs and a node. In this case there is only one node, so each input has a single synapse connection. You can think of these as the inputs to a function, just like in math where we have a function, say f that takes an input x and returns a value y that is a result of an operation on x. In this case x is the inputs and y is the part of the value generated by the node. The synaptic weights are essentially values that tell the function f how much it matters to the final output. These weights are initially random values between -1 and 1, they are adjusted through each training iteration and change how much the inputs matter the result. Representing this mathematically, you can think of it as such:

$$y = f(x) * \ w$$

Where, y is result, f(x) is some arbitrary function and w is the synaptic weighting.

This is how the network learns, much like humans by trying different approaches and then ranking them by how close they were to the result that is desired. Generally, this is called reinforcement learning, where a “reward” is given when the network gets closer to the output (increasing weight) and a “punishment” for when it gets further away (decreasing weight).

Thirdly, one of the most important parts is the summing junction or node. This is responsible for taking the values given to it from the synapses and turns it into a usable value. Achieving this is quite easy in this case, just requiring a sum over all of the input values:

$$\sum_{i=0}^{m} x_i \times w_i$$

Essentially what this does it is generates a vector of values that represent how well the network’s “guess” for the input was. This can also be represented as the dot product between the input vector or matrix and the vector that represents the weights:

$$M_{inputs}^{T} \cdot V_{weights}$$

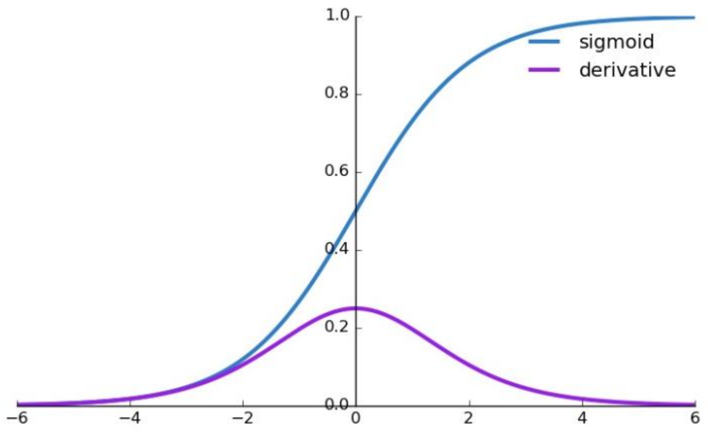
Note that the inputs are transposed, this is to make the columns and rows match up with the correct values in the weights vector. If you are familiar with the dot product then you would see that this is exactly the same as the weighted sum from above, but just applied between the columns of the input matrix and the entries of the weights vector. If the mathematical representation was a bit confusing, then it may be helpful to think of this as water streams. Imagine you have multiple streams of different colours that eventually converge at a point to create a single colour. These streams are the input values from the synapses (the values in the input matrix). For each of the water streams there are dams or gates that you can open and close to different amounts, letting different amounts of the coloured water through. These are the weightings, depending on how open or closed a gate is affects how much that coloured stream changes the final colour of all the streams merging together.

Next on the list is the activation function. One of the most important parts of neural networks is an idea called back propagation. Essentially, what this does is gather the values that the summing junction generated and compare them to the weights. It is called an activation function because it uses a function (let’s say g) that takes a value and returns a value depending on how positive or negative it is. There are lots of these activation functions, one of the most common (and easy to understand) is the sigmoid function and its derivative:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{d\sigma(x)}{dx} = \left(\frac{1}{1+e^{-x}}\right)\left(\frac{-e^{-x}}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x))$$

When graphed, it looks like this:



As you can see, it limits the input values between 0 and 1. Something to note is that the only way to get a resulting value of 0 or 1 is to have an input value of $\pm\infty$, which is obviously not going to happen since it's not possible to do infinitely many things in a finite amount of time. Okay, back on track now. Our friend the sigmoid here is responsible for telling the network how much the weights of synapses realistically need to change, actually it's the derivation of the sigmoid that does this. If you are familiar with what a derivative it, its essentially how much the output changes as you move change the input value, otherwise known as the slope or delta. For example, imagine a synapse has a weight of 50, this implies that it is very likely (basically always) to be used with an input value. But, something to note is that the weights of the synapses at the very start of training were between -1 and 1, 50 is pretty far away from either of those values. It is essentially a guarantee at that point, so increase that weight more and more with greater values seems a bit ridiculous and unnecessary. The sigmoid basically caps the value with a small delta as it gets large or small so you don't end up with unnecessarily large value, potentially skewing your results.

Continuing with the coloured stream analogy from before, you can imagine that there is a point where opening a dam or gate beyond where it current is doesn't really make a difference. So, the sigmoid essentially tells the gate "you don't need to open this much more because it's not going to make any significant difference". So, the gate may only open a tiny bit more. It is important to note that it doesn't stop the value changing, it just limits it. The reason being that stopping it can affect the results.

There are lots of activation functions, and depending on the data you have and also what type of results you want, accuracy and training type, you might use a different function. Choosing an activation function can be fairly difficult, it is important to recognise that it can also affect your training time given that more complicated functions take longer to compute. Albeit this difference is not huge, but definitely can be measurable if you are too carried away with finding the perfect function. Keeping it as simple as possible but not sacrificing accuracy is essential. There isn't really any standards or definite outlines for defining functions, there is a lot of content about when you should use different types of functions, but everyone has different training data so experimenting with different functions is sometimes the best method. I would encourage you to have a look into what different types of functions actually do and when they are used. Just to give you an idea, here are some other functions (excluding sigmoid):

- Binary step
  - $f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$
- Inverse square root unit
  - $f(x) = \frac{x}{\sqrt{1+\alpha x^2}}$
- Rectified linear unit
  - $f(x) = \begin{cases} 0 \; for \; x \leq 0 \\ x \; for \; x > 0 \end{cases}$
- Gaussian
  - $f(x) = e^{-x^2}$
- Soft clipping
  - $f(\alpha, x) = \begin{cases} -\frac{\ln(1-\alpha(x+\alpha))}{\alpha} & for \; \alpha < 0 \\ x & for \; \alpha = 0 \\ \frac{e^{\alpha x}-1}{\alpha} + \alpha & for \; \alpha > 0 \end{cases}$

It's all well and good to talk about the theory of neural networks, but it's also useful to see it in action. Let's take a look at an example of the perceptron model build in python 3.8.0. We are going to need the Numpy library to work with vectors and matrices, if you haven't got it installed head to terminal or command prompt and install it with: "*pip install numpy*"

First off, let's import Numpy and create a new class and a constructor so we can set up our synaptic weights. Python's class structure allows this to be done with the __init__ dunder/info method.

```python
import numpy as np

class NN:

    def __init__(self, seed=1):
        np.random.seed(seed)
        # Give the synapses random weightings initially
        self.synaptic_weights = 2 * np.random.random((3, 1)) - 1
```

You might notice that there is a seed argument in the constructor function. This is not really necessary, but it helps when testing the network to make it more deterministic. Moving on from that, what are we actually doing here? Essentially, we are generating a 3x1 vector of random values between -1 and 1. The np.random.random function generates a random float between 0 and 1, so we multiple that value by 2 and then minus 1, keeping it in the range of -1 to 1.

You might be confused as to why such specific dimensions, this has to do with the training data used in this example. The data is a matrix with dimensions $3 \times n$ (n is just some value indicating how many training cases you are using, feel free to change it as you wish, keep in mind it must match the data you are using.

So, we have the weightings randomly generated and have some control of what the weightings are initially, since our seed is 1 they will always be the same values. Next, let's define the activation function and its derivative since that it going to be important for later. In this example, I'm going to be using the sigmoid function since we discussed it earlier. Recalling the function and its derivative:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Let's translate this into some code, specifically into data methods (methods with two underscores at the start of the method name, e.g. __test_func()).

```
def __sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

def __sigmoid_derivative(self, x):
        return self. __sigmoid(x) * (1 - self. __sigmoid(x))
```

Alrighty, now that we have the necessary functions and data structures, we can implement the methods for training the network and thinking/guessing an output. I'm going to split this into two separate functions; train and think. The reason behind this is that it would be good to be able to access the think function later on to consider some set of inputs after we have trained the network. Firstly, the train function needs to have three inputs, the training data inputs, outputs and the number of iterations (number of times the network analyses the input data and refines its weightings and guesses). We can define the train function as follows:

```
def train(self, t_inputs, t_outputs, iters):
        for iterations in range(iters):
            # Calculate what the output could be given current
              understanding (weightings)
            output = self.think(t_inputs)
            # See how much the guess differs from the actual answer
            error = t_outputs - output
            # Improve understanding of the data by changing the weighting
              values of synapses
            self.synaptic_weights += np.dot(t_inputs.T, error *
                                        self.__sigmoid_derivative(output))
```

Next, let's define the think function. This one is substantially easier, essentially all it does it takes the weighted sum of the input values and the weights then passes it through the sigmoid function.

```
def think(self, inputs):
        # Calculate a guessed output as a product of the inputs and
          weights
        return self.__sigmoid(np.dot(inputs, self.synaptic_weights))
```

Alrighty! That's just about it, so let's define some training data and give it a go. I'm going to use a python convention to make the file runnable. To do this you just need one statement where all your runnable goes: if __name__ == "__main__":

Inside here we can define the training data, instantiate a new network and get it to think about some additional situations. Let’s go ahead and do that.

```
if __name__ == "__main__":

    TRAINING_ITERATIONS = 100000

    neural_network = NN()
    print("Random starting weights:\n", neural_network.synaptic_weights)

    training_inputs = np.array([[0, 0, 1],
                                [1, 1, 1],
                                [1, 0, 1],
                                [0, 1, 1],
                                [0, 1, 0],
                                [0, 0, 0]])

    training_outputs = np.array([[0, 1, 1, 0, 0, 0]]).T

    neural_network.train(training_inputs, training_outputs, TRAINING_ITERATIONS)
    print("Post-training weights:\n", neural_network.synaptic_weights)
    print("Considering new situation [1, 0, 0]:\n", neural_network.think(np.array([1, 0, 0])))
```

Let’s take a quick look at the training data in a bit more detail.

Inputs			Output
0	0	1	0
1	1	1	1
1	0	1	1
0	1	1	0
0	1	0	0
0	0	0	0

So, what we have is a few cases of 3 bits. You might notice that the only time the output is a 1 is when the first digit is a 1, vice versa for zero as well. This is a very simple set of data just mean to illustrate the point, but you could expand this as much as you want, as long as the dimensions of the data match. A good way to think about it is that each row of the inputs is a 2D to 1D flatten input of something like an image. In this case it’s a very simple image, and what we are looking for are images where the left bit (which could be the top pixel) is filled. You can image how this could be scaled and there are many other uses and examples for data that you could implement.

Enough talk, let’s run the network and see what it says for the case [1, 0, 0]!
Running it we get this as the output:

```
Random starting weights:
 [[-0.16595599059485]
 [0.44064898688431]
 [-0.99977125036531]]
Post-training weights:
 [[26.270731592876633]
 [-8.741166324004865]
 [-8.74100636065231]]
Considering new situation [1, 0, 0]:
 [0.9999999999961027]
```

After it did 100,000 training iterations, we asked it to tell us whether [1, 0, 0] is a 1 or a 0. It says [0.9999999999961027]. Hmm… well this is actually a great answer, what this means is that using the weights that is refined during training it decided that the number associated as an output for this input is arbitrarily close to 1. This is correct as the first number in the input vector is 1, if we think back to our talk of activation functions you might remember the part where we talked about the sigmoid function only ever producing values of absolute magnitude, 1 or 0, when you have infinitely many training iterations. So, with that in mind, this is essentially equivalent to 1.

Success! Our network accurately predicts an answer to a given input after training it, it’s worthwhile playing around with different training data, iterations and activation functions. You can get some interesting results and different learning types.