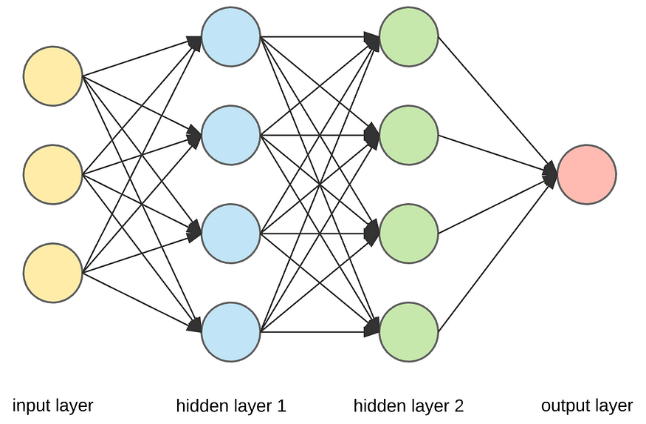
# An Introduction to Neural Networks

To many, including myself, Neural Networks can be difficult to wrap your head around. In Python, one of the most popular Machine Learning languages, there are many different libraries, of inbuilt classes and objects capable of simplifying the creation of neural networks.

In this document, we will aim to analyse and cover the basics of machine learning within Python using several example programs. I would also recommend checking out [3Blue1Brown’s video series](https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi) on Neural networks if you want a deeper look at the mechanisms behind these complicated systems.

#### The structure of a neural network

A neural network is a large collection of interconnected functions, known as **neurons**, that perform algorithms on given data. Each neuron is responsible for one individual algorithm and will pass on the result deeper into the network.

Neural networks consist of three main sections of neuron layers, the input layer, the hidden layer, and the output layer.

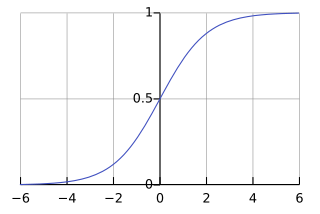
The input layer is responsible for data put into the system. The more complex the data, the more input layer neurons are required. For example, our first program only requires 3 input layer neurons, one for each digit in the XNOR gate input, while an image may require thousands of input layer pixels. These data

The hidden layer is the “brain” of the neural network where all the calculations are performed on the input data to create the output data. There will likely be more than one layer where calculations are performed, and the more calculations, the more complicated the code will be, and the longer the program will run, however the more complex the program will be.

The output layer is the display, that is, what the user receives. For chatbots, this is a series of 1s and 0s which make up the ASCII symbols for the text given back to the user. For image generators, these are also 1s and 0s representing information about the pixels displayed to the user in binary, such as the HEX code and position.

At the beginning of the program, when the class is initialised down at the end of the program, the input layer size, output layer size, and hidden layer sizes are all set to be 3, 1, and 4, where the size is the number of *neurons*.

#### The sigmoid (not sigma) function

The sigmoid function is very important in the creation of neural networks. Its graph can be seen on the right, and as you may notice, has a range between -1 and 1 for any real number, whether that be 4, -10, or 91 023. As the domain, or x-values, approach ∞, the range, or y-value, approaches 1, and as the domain approaches - ∞, the range approaches -1 but never reaches it. This allows non-linearity to be incorporated into the nodes of a feed-forward machine learning model, enabling the model to learn complex decision functions. Other functions, such as sine and cosine, which you should be familiar with from trigonometry, can be used because of the limitations to their range, however, due to their periodic (repeating) nature, they are not used. The sigmoid function is useful as larger, negative numbers will map to larger, negative numbers and larger, positive numbers will map to larger positive numbers. This allows for the manipulation of input and output values within a network.

## 2 Layer Neural Network

### Introduction

This file uses the [NumPy module](https://numpy.org/doc/stable/user/absolute_beginners.html) to work with multi-dimensional arrays, which display the program's inputs and outputs, to predict the result of an [XNOR gate](https://www.geeksforgeeks.org/xnor-gate/) input. (An XNOR gate is the combination of an [XOR gate](https://www.geeksforgeeks.org/xor-gate/) and a [NOT gate](https://www.geeksforgeeks.org/not-gate/) (Minecraft Redstone Reference!). An XNOR gate will give a True output (1) when all the inputs are the same, i.e. all True or all False, and a False output (0) otherwise, when the values are different.

### Line by Line

The program works using two sets of arrays, X and y. The X array is the array of possible inputs to the system, 3 True or False values stored as floats, and the y array has the system outputs for each input, both declared using NumPy’s array function.

import numpy as np

X = np.array(([0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]), dtype=float)

y = np.array(([1], [0], [0], [0], [0], [0], [0], [1]), dtype=float)

The xPredicted array is the input layer of the system and the predicted output should hence be 0.

xPredicted = np.array(([0, 0, 1]), dtype=float)

For this program, a Neural Network class is defined to ensure good code practice and allow for abstraction, as the final main program loop will thus be simple, only a series of functions within a for loop where the source code is hidden behind the function and class names.

class Neural\_Network (object):

Here, during the creation of the neural network class, we initialise the size of the neural network’s layers, with the input layer having 3 nodes, the output layer having 1 node, and the hidden layer having 4 nodes.

    def \_\_init\_\_(self):

        self.inputLayerSize = 3

        self.outputLayerSize = 1

        self.hiddenLayerSize = 4

The program then connects the input layer and hidden layers together using W1, a randomly generated weight matrix, and the hidden layer to the output layer using W2, another randomly generated weight matrix.

        self.W1 = np.random.randn(self.inputLayerSize, self.hiddenLayerSize)

        self.W2 = np.random.randn(self.hiddenLayerSize, self.outputLayerSize)

We define a loss file that will store the values so we can graph them.

        self.lossFile = open("SumSquaredLossList.csv", "a")

The feedForward function takes in X, the input array, and passes it through the four hidden layers. z, the weighted sum, is equal to the matrix multiplication of X and W1, known as the [dot product](https://www.youtube.com/watch?v=Tfuu7iSxGIA). z2 is output when the weighted sum is passed through the sigmoid function, and thus is a value between -1 and 1. z3 is the weighted sum of z2 and W2, and the final output o is the result of passing the weighted sum z3 through the sigmoid function a final time.

    def feedForward(self, X):

        self.z = np.dot(X, self.W1)

        self.z2 = self.activationSigmoid(self.z)

        self.z3 = np.dot(self.z2, self.W2)

        o = self.activationSigmoid(self.z3)

        return o

The backwardPropagate function is used to minimise the error of o when compared to y. y is the target output of the system, and the error is the result of subtracting o from y. The purpose of this entire algorithm is to reduce the error. o\_delta is the product of the derivative of the sigmoid function and the error, which can be used for adjustment. The next two lines determine how much the hidden layer weights contributed to the overall error and generate an adjustment value, and then W1 and W2 are adjusted based on this data.

    def backwardPropagate(self, X, y, o):

        self.o\_error = y - o

        self.o\_delta = self.o\_error \* self.activationSigmoidPrime(o)

        self.z2\_error = self.o\_delta.dot(self.W2.T)

        self.z2\_delta = self.z2\_error \* self.activationSigmoidPrime(self.z2)

        self.W1 += X.T.dot(self.z2\_delta)

        self.W2 += self.z2.T.dot(self.o\_delta)

This is the main function loop, and it is the combination of the feedforward and backwardPropagate functions.

    def trainNetwork(self, X, y):

        o = self.feedForward(X)

        self.backwardPropagate(X, y, o)

The following two functions create the sigmoid and the first derivative of the sigmoid function.

    def activationSigmoid(self, s):

        return 1/(1+np.exp(-s))

    def activationSigmoidPrime(self, s):

        return s \* (1 - s)

    def saveSumSquaredLossList(self, i, error):

        self.lossFile.write(str(i) + ", " + str(error.tolist())+"\n")

        self.lossFile.flush()

### Full Program

import numpy as np

X = np.array(([0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]), dtype=float)

y = np.array(([1], [0], [0], [0], [0], [0], [0], [1]), dtype=float)

xPredicted = np.array(([0, 0, 1]), dtype=float)

X = X/np.amax(X, axis=0)

xPredicted = xPredicted/np.amax(xPredicted, axis=0)

class Neural\_Network (object):

    def \_\_init\_\_(self):

        self.inputLayerSize = 3

        self.outputLayerSize = 1

        self.hiddenLayerSize = 4

        self.W1 = np.random.randn(self.inputLayerSize, self.hiddenLayerSize)

        self.W2 = np.random.randn(self.hiddenLayerSize, self.outputLayerSize)

        self.lossFile = open("SumSquaredLossList.csv", "a")

    def feedForward(self, X):

        self.z = np.dot(X, self.W1)

        self.z2 = self.activationSigmoid(self.z)

        self.z3 = np.dot(self.z2, self.W2)

        o = self.activationSigmoid(self.z3)

        return o

    def backwardPropagate(self, X, y, o):

        self.o\_error = y - o

        self.o\_delta = self.o\_error \* self.activationSigmoidPrime(o)

        self.z2\_error = self.o\_delta.dot(self.W2.T)

        self.z2\_delta = self.z2\_error \* self.activationSigmoidPrime(self.z2)

        self.W1 += X.T.dot(self.z2\_delta)

        self.W2 += self.z2.T.dot(self.o\_delta)

    def trainNetwork(self, X, y):

        o = self.feedForward(X)

        self.backwardPropagate(X, y, o)

    def activationSigmoid(self, s):

        return 1/(1+np.exp(-s))

    def activationSigmoidPrime(self, s):

        return s \* (1 - s)

    def saveSumSquaredLossList(self, i, error):

        self.lossFile.write(str(i) + ", " + str(error.tolist())+"\n")

        self.lossFile.flush()

    def saveWeights(self):

        np.savetxt("weightsLayer1.txt", self.W1, fmt = "%s")

        np.savetxt("weightsLayer2.txt", self.W2, fmt = "%s")

    def predictOutput(self):

        print("Predicted XOR output data based on trained weights: ")

        print("Expected (X1 - X3): \n" + str(xPredicted))

        print("Output (Y1): \n" + str(self.feedForward(xPredicted)))

    def closeFile(self):

        self.lossFile.close()

myNeuralNetwork = Neural\_Network()

trainingEpochs = 1000000

for i in range(trainingEpochs):

    print("Epoch # " + str(i) + "\n")

    print("Network Input: \n" + str(X))

    print("Expected Output of XOR Gate Neural Network: \n" + str(y))

    print("Actual Output from XOR Gate Neural Network: \n" + str(myNeuralNetwork.feedForward(X)))

    Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))

    myNeuralNetwork.saveSumSquaredLossList(i, Loss)

    print("Sum Squared Loss: \n" + str(Loss))

    print("\n")

    myNeuralNetwork.trainNetwork(X, y)

myNeuralNetwork.saveWeights()

myNeuralNetwork.predictOutput()

myNeuralNetwork.closeFile()

## Tensor Flow Keras Network