Building Simple Neural Network Using Numpy

To gain better understanding and intuition about Neural Network is to build a Nueral Net from Scratch with out using high end libraries like Tensorflow, Keras, theano etc.The reason for it is because it abstract or hides the inner working of maths involved in the Neural net. so lets talk about what Neural Net is. Most introductory info describes the working of Neural Net as brain. like every neuron in the human brain is connected with other neurons for communication and performing different tasks of the human body. Similary the Neural Network nodes is also connected to one an other for effective communication and producing better outputs based on the inputs. In mathematics we can describe Neural Network as a mathematical function that maps a given input to a desired output.

Neural Networks consist of the following components An input layer, X

An arbitrary amount of hidden layers

An output layer, Y

A set of weights and biases between each layer, W and b

A choice of activation function for each hidden layer, σ. In this tutorial, we'll use a Sigmoid activation function.

Below is a diagram of two layer architecture.

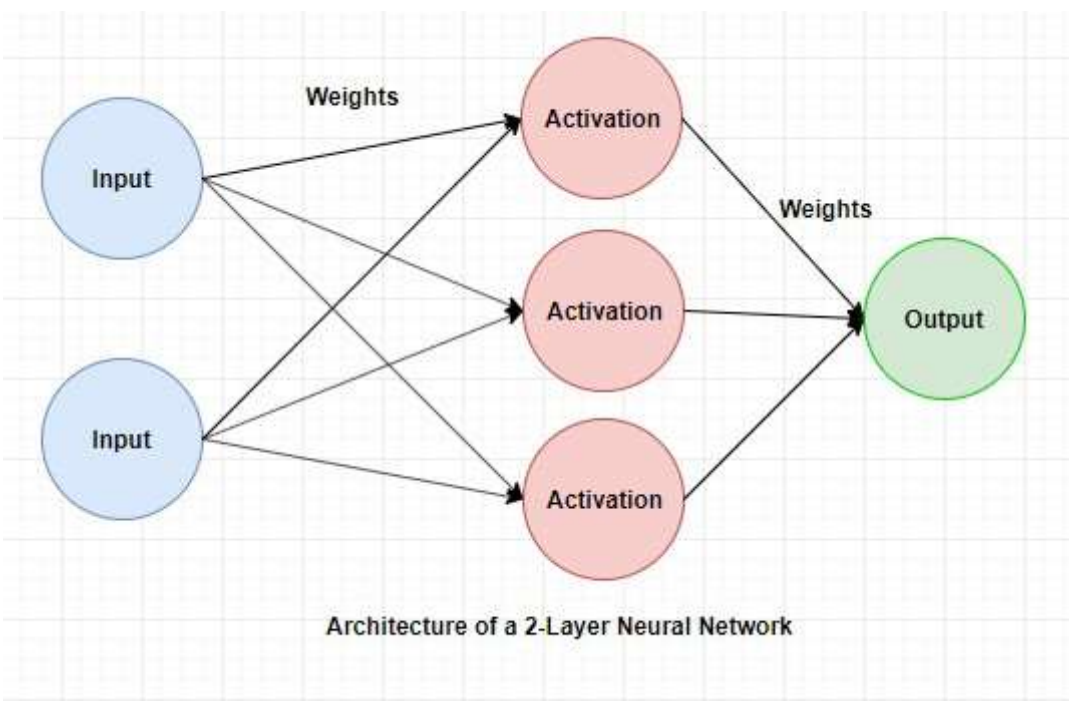

Figure no #1

In [137]:
```
import numpy as np
```

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. It has very useful mathematical methods which will help us in completing this task.

Neural networks work in very similar manner. It takes several input in our case 2 inputs, processes it through multiple neurons from multiple hidden layers and returns the result using an output layer. This result estimation process is technically known as FORWARD PROPAGATION.

Next, we compare the result with actual output. The task is to make the output to neural network as close to actual (desired) output. Each of these neurons are contributing some error to final output. How do you reduce the error?

We try to minimize the value/ weight of neurons those are contributing more to the error and this happens while traveling back to the neurons of the neural network and finding where the error lies. This process is known as BACK PROPAGATION. In order to minimize the error, the neural networks use a common algorithm known as "Gradient Descent".

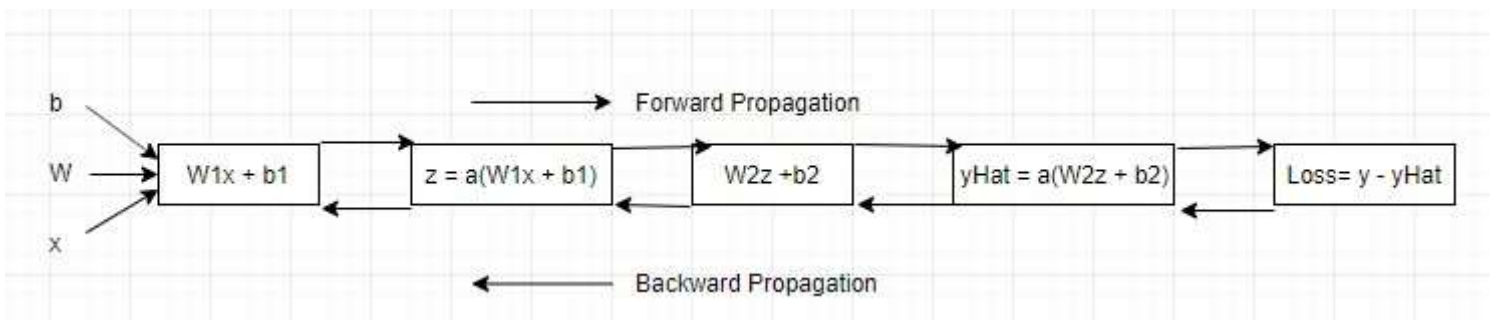

Figure no #2

So lets start coding by making the class of our Neural Network. So the first method is to build the constructor. Here we will initialize the inputLayerSize, hiddenLayerSize, outputLayerSize and the weights W1 and W2 and making baises to 0.

According to above diagram we have two inputs, one hidden layer three activations and one output layer.

Forward Propagation:

Step 1: We have to model the relationship between two variables by fitting a linear equation to observed data. which is Y= ax + b. here we have weight (W) instead (a) so the equation becomes y=Wx +b. In this step we take matrix dot product of input and weights assigned to edges between the input and hidden layer then add biases of the hidden layer neurons to respective inputs, this is known as linear transformation.

Step 2: Perform non-linear transformation using an activation function (Sigmoid) on the data obtained from previous linear equation task restrict and classify the values between 0 and 1 and also used as output activation function for binary classification problems. Sigmoid will return the output as 1/(1 + exp(-x)).

Step 3: Perform a linear transformation on hidden layer activation (take matrix dot product with weights and add a bias of the output layer neuron) then apply an activation function (again used sigmoid, but you can use any other activation function depending upon your task) to predict the output.

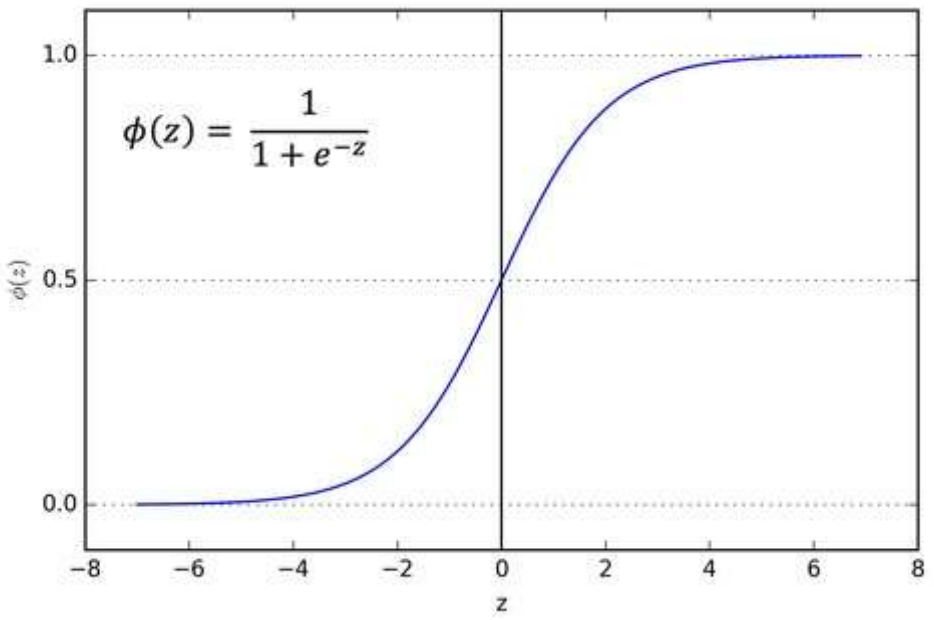

Figure no #3

In [138]:
```
class Neural_Network(object):
    def __init__(self):
        self.inputLayerSize=2  #intilialize Input, Hidden and Output layers
        self.outputLayerSize=1
        self.hiddenLayerSize=3


        self.W1=np.random.randn(self.inputLayerSize,self.hiddenLayerSize)
        self.W2=np.random.randn(self.hiddenLayerSize,self.outputLayerSize)
        #initialize waits using np.random.randn() which creates an array of specified shape and fills it with
random values
        #as per standard normal distribution.


    def sigmoid(self,z):         #Sigmoid Function
        return 1/(1+np.exp(-z))

    def forward(self, X):
        self.z1=np.dot(X,self.W1)    # will do matrix multiplication eg:Matrix
                                        #a:
                                        #[[1 2]
                                       # [3 4]]
                                       #Matrix b:
                                        #[[ 5 10]
                                        # [15 20]]
                                       #Resultant Matrix:
                                      # [[ 35  50]
                                        #[ 75 110]]
        self.a1=self.sigmoid(self.z1)  #restricting values values between 0 and 1
        self.z2=np.dot(self
                       .a1,self.W2) #again matrix multiplication using prev values of a2 and W2
        yHat=self.sigmoid(self.z2)  # sigmoid on z3 which have values from a2 and W2
        return yHat

    def costFunction(self, X,y):   #calculate error between actual and predicted values
        self.yHat=self.forward(X)
        J=0.5*sum((y-self.yHat)**2)
        return J

    def sigmoidPrime(self,z):
        return np.exp(-z)/((1+np.exp(-z))**2) #derivative of Sigmoid funtion

    def costFunctionPrime(self,X,y):   #Applying chain rule and partial derivative to get the derivatives of
 each layer in NN
        self.yHat=self.forward(X)
        delta3=np.multiply(-(y-self.yHat),self.sigmoidPrime(self.z2))  #Using Equation 5 and Equation 6 in Fi
gure no #5
        djdW2=np.dot(self.a1.T,delta3)                                   #for gradient of output layers weights

        delta2=np.dot(delta3,self.W2.T)*self.sigmoidPrime(self.z1)   # Using Equation 11 in Figure no #7 for
 finding
        djdW1=np.dot(X.T,delta2)                                        #the gradient of hidden layer weights

        return djdW1,djdW2
```

Step 4: After Forward Propogation we have to Compare prediction with actual output and calculate the gradient of error (Actual – Predicted). Error is the mean square loss = ((Y-yHat)^2)/2

Step 5: Now that we've measured the error of our prediction (loss), we need to find a way to propagate the error back, and to update our weights and biases. In order to know the appropriate amount to adjust the weights and biases by, we need to know the derivative of the loss function with respect to the weights and biases in our case biases are zero.

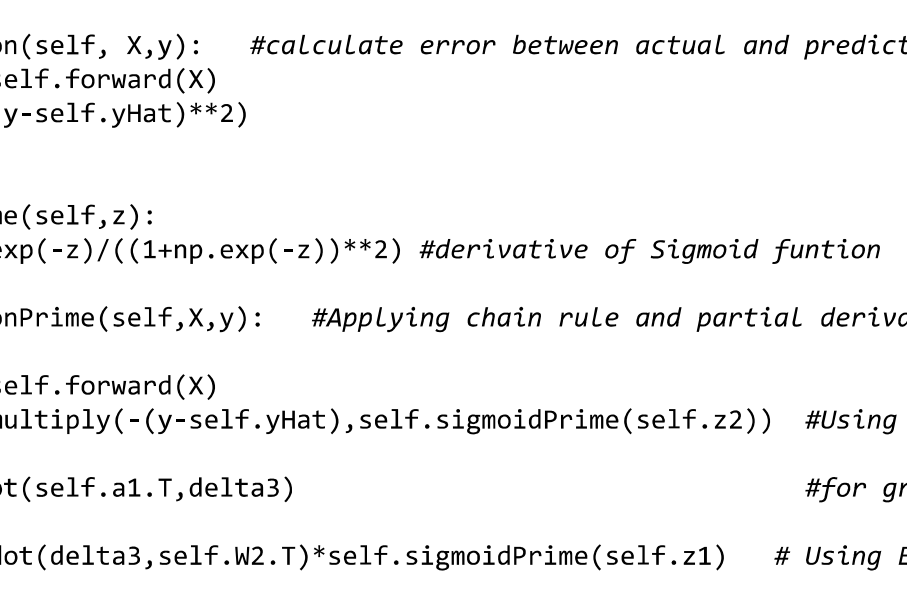

Figure no #4

Gradients for Output Layer Weights:

Output layer connection weights, wjk

Since the output layer parameters directly affect the value of the error function, determining the gradients for those parameters is fairly straight-forward.

$$E = \frac{1}{2}\sum_{k\in K}(a_k - t_k)^2$$ Equation 1

$$\begin{aligned}\frac{\partial E}{\partial w_{jk}} &= \frac{1}{2}\sum_{k\in K}(a_k - t_k)^2 \\ &= (a_k - t_k)\frac{\partial}{\partial w_{jk}}(a_k - t_k)\end{aligned}$$ Equation 2

$$\begin{aligned}\frac{\partial E}{\partial w_{jk}} &= (a_k - t_k)\frac{\partial}{\partial w_{jk}}a_k \\ &= (a_k - t_k)\frac{\partial}{\partial w_{jk}}g_k(z_k) \\ &= (a_k - t_k)g'_k(z_k)\frac{\partial}{\partial w_{jk}}z_k,\end{aligned}$$ Equation 3

$$\frac{\partial E}{\partial w_{jk}} = (a_k - t_k)g'_k(z_k)a_j$$ Equation 4

$$\delta_k = (a_k - t_k)g'_k(z_k)$$ Equation 5

$$\frac{\partial E}{\partial w_{jk}} = \delta_k a_j$$ Equation 6

Figure no #5

Gradients for Hidden Layer Weights

Due to the indirect affect of the hidden layer on the output error, calculating the gradients for the hidden layer weights (wij) is somewhat more involved. However, the process starts just the same:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{1}{2}\sum_{k\in K}(a_k - t_k)^2 \\ &= \sum_{k\in K}(a_k - t_k)\frac{\partial}{\partial w_{ij}}a_k\end{aligned}$$ Starts with Equation 2

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \sum_{k\in K}(a_k - t_k)\frac{\partial}{\partial w_{ij}}g_k(z_k) \\ &= \sum_{k\in K}(a_k - t_k)g'_k(z_k)\frac{\partial}{\partial w_{ij}}z_k\end{aligned}$$ Equation 7

$$\begin{aligned}z_k &= b_k + \sum_j a_j w_{jk} \\ &= b_k + \sum_j g_j(z_j)w_{jk} \\ &= b_k + \sum_i g_j(b_i + \sum_i z_i w_{ij})w_{jk}\end{aligned}$$ Equation 8

$$\begin{aligned}\frac{\partial z_k}{\partial w_{ij}} &= \frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \\ &= \frac{\partial}{\partial a_j}a_j w_{jk}\frac{\partial a_j}{\partial w_{ij}} \\ &= w_{jk}\frac{\partial a_j}{\partial w_{ij}} \\ &= w_{jk}\frac{\partial g_j(z_j)}{\partial w_{ij}} \\ &= w_{jk}g'_j(z_j)\frac{\partial z_j}{\partial w_{ij}} \\ &= w_{jk}g'_j(z_j)\frac{\partial}{\partial w_{ij}}(b_i + \sum_i a_i w_{ij}) \\ &= w_{jk}g'_j(z_j)a_i\end{aligned}$$ Equation 9

Figure no #6

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \sum_{k\in K}(a_k - t_k)g'_k(z_k)w_{jk}g'_j(z_j)a_i \\ &= g'_j(z_j)a_i \sum_{k\in K}(a_k - t_k)g'_k(z_k)w_{jk} \\ &= a_i g'_j(z_j)\sum_{k\in K}\delta_k w_{jk}\end{aligned}$$

Equation 10

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= a_i g'_j(z_j)\sum_{k\in K}\delta_k w_{jk} \\ &= \delta_j a_i\end{aligned}$$

Equation 11

where

$$\delta_j = g'_j(z_j)\sum_{k\in K}\delta_k w_{jk}$$

Figure no #7

In order to get better understaning of the derivation i put the link where there is detailed info about the derivatives of each layer step by step
https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/ (https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/)

In [139]:
```
X=np.array([[0,0],[1,0],[0,1],[1,1]])           #input array of X
```

In [140]:
```
y=np.array([0,1,1,1])  #output actual values
```

In [141]:
```
X.shape
```

Out[141]: (4, 2)

In [142]:
```
y.shape
```

Out[142]: (4,)

Here i will train model for OR gate operation where A HIGH output (1) results if one or both the inputs to the gate are HIGH (1). If neither input is high, a LOW output (0) results.

here i will get random values for X between 0 and 1 where y is the addition of operand or element of X

In [143]:
```
np.random.seed(4)
X=np.random.rand(10000,2)    #getting random 10000 samples using np.random.rand() which output random values
 between 0 and 1
y=np.apply_along_axis(lambda element: element[0]+element[1],axis=1,arr=X)
#apply_along_axis will apply the lambda fucntion which is element wise addition of values using array X along
column axis
#y=np.random.rand(10000,1)
```

In [144]:
```
X.shape
```

Out[144]: (10000, 2)

In [145]:
```
y.shape
```

Out[145]: (10000,)

In [146]:
```
X
```

Out[146]:
```
array([[ 0.96702984,  0.54723225],
       [ 0.97268436,  0.71481599],
       [ 0.69772882,  0.2160895 ],
       ...,
       [ 0.63652491,  0.84404444],
       [ 0.01889563,  0.3513995 ],
       [ 0.0708804 ,  0.09866774]])
```

In [147]:
```
y
```

Out[147]:
```
array([ 1.51426209,  1.68750035,  0.91381832, ...,  1.48056935,
        0.37029513,  0.16954813])
```

In [148]:
```
y=y.reshape(10000,1) #rehshape the array in to vector array
```

In [149]:
```
y.shape
y
```

Out[149]:
```
array([[ 1.51426209],
       [ 1.68750035],
       [ 0.91381832],
       ...,
       [ 1.48056935],
       [ 0.37029513],
       [ 0.16954813]])
```

In [150]:
```
#X=np.linalg.norm(X)
#y=np.linalg.norm(y)
```

In [151]:
```
NN=Neural_Network()       #creating an object of Neural Network class
max_iteration=10000        #no of iteration
iter=0
learningRate=0.01        #learning rate its like taking steps toward local optima.  it should not be maximum
 nor minimum
while iter<max_iteration:
    djdW1, djdW2 = NN.costFunctionPrime(X,y)    #return the trained parameters and assign to djdW1, djdW2

    NN.W1=NN.W1-learningRate*djdW1    #update Parameter W1 and W2 based on djdW1 and djdW2
    NN.W2=NN.W2-learningRate*djdW2

    if iter%1000==0:
        print (NN.costFunction(X,y))  #print the costfucntion after every 1000 iteration
    iter=iter+1
```

```
[ 955.62804742]
[ 470.59807765]
[ 458.17174786]
[ 456.84621598]
[ 453.21727266]
[ 463.60520336]
[ 473.39717028]
[ 529.75791616]
[ 581.31544319]
[ 448.99298218]
```

In [152]:
```
print (y)
print("---")
print (NN.forward(X))  # here we just check how our model perform by comparing actual values and the predicte
d ones.
```

```
[[ 1.51426209]
 [ 1.68750035]
 [ 0.91381832]
 ...,
 [ 1.48056935]
 [ 0.37029513]
 [ 0.16954813]]
---
[[ 0.99490012]
 [ 0.99693468]
 [ 0.94275001]
 ...,
 [ 0.99545428]
 [ 0.25156713]
 [ 0.05233312]]
```

sse is same function as we used for cost function it is the sum of the squared differences between each actual and predicted observations. here i just used to see if the model is working well and how much error we have in our predictions.

In [153]:
```
def sse(y,yHat):
    return sum(map(lambda ab : pow(ab[0]-ab[1],2), zip(y,yHat)))
#print(sse(y,NN.forward(X)))
print(sse(y,NN.forward(X))/10000)
```

```
[ 0.08943408]
```

In [154]:
```
pred=np.random.rand(1,2)
print(pred)               #lets take random sample check the difference between the actual and predicted valu
e by the model
                            #the predicted result is quite close to the actual value if u run this code snipp
et
result=NN.forward(pred)
print(result)
```

```
[[ 0.34941852  0.30649525]]
[[ 0.77974212]]
```

lets create an other object Neural Network Class and use the Sum of squared error method. this time we didnt train the model and the SSE will produce greater error than the output we get from trained model.

In [155]:
```
NN=Neural_Network()
```

In [156]:
```
#print(sse(y,NN.forward(X)))

print(sse(y,NN.forward(X))/10000)
```

```
[ 0.36054183]
```