4 : Artificial Neural Network, Backpropagation and testing

```
import numpy as np

'''
    each sample in dataset has
'''
        # [ hours_of_sleeping, hours_of_studying,  marks_obtained_in_a_test ]


dataset = [  [2, 9, 92],
             [1, 5, 86],
             [3, 6, 89]
          ]
```

Preparing inputs and outputs

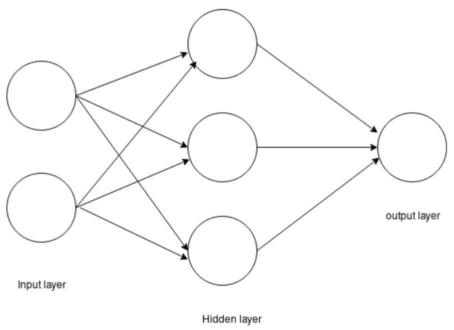
```
# Forming inputs and outputs for the Neural Network

inputs  = [ sample[:-1]      for sample in dataset ]
outputs = [ [ sample[-1] ]   for sample in dataset ]

            # Notice : extra big brackets for generating column array, highly critical if it is ignored


# converting normal arrays to numpy float arrays,

inputs = np.array(inputs, dtype=float)
outputs =  np.array(outputs, dtype=float)


# Normalizing by max divide, The inputs and outputs shoud be in range [0, 1]

inputs = inputs/np.amax(inputs, axis=0)     # input has 2 columns, dividing each column with maximum of respective column
outputs = outputs/100                       # since max test score is 100
```

Defining necessary functions

```
#activation function

def sigmoid(s):

    return 1/(1 + np.exp( -s ) )
```

```
#dervative of activation function which is necessary for backtracking

def sigmoid_derivative(s):

        return s * (1 - s)
```

static single hidden layered Neural Net class without biases



```python
class Neural_Network(object):


    def __init__(self, inputs, outputs):

                   # Initialization
        self.inputs = inputs
        self.outputs = outputs

        #--------------------------------------------------------

                   # Parameters
        self.input_size  = len(inputs[0])
        self.output_size = 1
        self.hidden_size = 3                   # number of neurons in hidden layer

        #--------------------------------------------------------

                                               # random weights for first layer and second layer'


        self.weights1 = np.random.randn(self.input_size, self.hidden_size)        # (3x2) weight matrix from input to hidden layer
        self.weights2 = np.random.randn(self.hidden_size, self.output_size)       # (3x1) weight matrix from hidden to output layer


    def forward(self, inputs):

        ''' forward propogation  '''


    #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++#

        self.z1 = np.dot(inputs, self.weights1)        # dot product of input layer and first set of 3x2 weights
        self.z2 = sigmoid(self.z1)                     # activation function applied on previous output

    #+-------------------------------------------------------------------------------------------------------+#

        self.z3 = np.dot(self.z2, self.weights2)       # dot product of hidden layer and second set of 3x1 weights
        obtained_output = sigmoid(self.z3)             # final activation function

    #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++#

        return obtained_output


    def backward(self, obtained_output):

        ''' backward propgate through the network    '''


                # error in output
        output_error = self.outputs - obtained_output

                # applying derivative of sigmoid to obtained outputs
        output_delta = output_error * sigmoid_derivative(obtained_output)

        '''---------------------------------------------------------------'''

                 # z2 error:  hidden layer weights contribribution to output error
        z2_error = output_delta.dot( self.weights2.transpose()    )

                 # applying derivative of sigmoid to z2 error
        z2_delta = z2_error * sigmoid_derivative(self.z2)


        '''---------------------------------------------------------------'''

        #----------------------------------------------------#
        #   updating weights, wj -> wj + n * delta(wj)       #
        #                               n, learning rate    #
        #----------------------------------------------------#

                  # adjusting first set (input --> hidden) weights
        self.weights1 = self.weights1 +   (0.5 * self.inputs.T.dot(z2_delta) )     #.T stands for transpose

                  # adjusting second set (hidden --> output) weights
        self.weights2 = self.weights2 +   (0.5 * self.z2.T.dot(output_delta) )


    def train(self):

        ''' training  the network'''

        obtained_output = self.forward(self.inputs)
        self.backward(obtained_output)
```

Creating a neural net and training it

```
net = Neural_Network(inputs, outputs)


for i in range(20):                   #  the more you train, the  less error you obtain untill it overfits

    loss =  np.mean( np.square(outputs - net.forward(inputs)))      # mean sum squared loss
    print ("Epoch-> ", i, " Loss:", loss)
    net.train()
```

```
Epoch->  0  Loss: 0.07274521673240818
Epoch->  1  Loss: 0.0585782542249422
Epoch->  2  Loss: 0.047897611723800394
Epoch->  3  Loss: 0.039725730089891075
Epoch->  4  Loss: 0.03337521083036012
Epoch->  5  Loss: 0.028364553631714123
Epoch->  6  Loss: 0.0243542932035502
Epoch->  7  Loss: 0.021102324867346034
Epoch->  8  Loss: 0.018433588767724624
Epoch->  9  Loss: 0.01621966047452225
Epoch->  10  Loss: 0.014364967246628403
Epoch->  11  Loss: 0.01279739659922404
Epoch->  12  Loss: 0.011461822793107186
Epoch->  13  Loss: 0.01031558692703971
Epoch->  14  Loss: 0.009325299066303581
Epoch->  15  Loss: 0.008464545890300382
Epoch->  16  Loss: 0.007712226436721825
Epoch->  17  Loss: 0.007051329051034897
Epoch->  18  Loss: 0.006468022117193933
Epoch->  19  Loss: 0.005950970628860717
```

Prediction

```
test = np.array( [ [2, 5], [1, 9], [2, 3] ], dtype=float)    # same as preparing inputs
test = test/np.amax(test, axis=0)

print("Input: ",test, sep='\n\n')                            # Normalized input
```

```
Input:

[[1.         0.55555556]
 [0.5        1.        ]
 [1.         0.33333333]]
```

```
output_of_test =  net.forward(test) * 100

print("Predicted Output", output_of_test, sep='\n\n')        # predicted marks for each sample
```

```
Predicted Output

[[86.19906654]
 [79.93228323]
 [86.8421938 ]]
```