```
import random
import numpy as np
import matplotlib.pyplot as plt
import math
from random import shuffle

#Number of points in two layers
N = 300

class Net:
    def __init__(self):
        self.X = self.findPoints(0, 1, N)
        self.V = self.findPoints(-1 / 10, 1 / 10, N)
        self.D = self.getActualOutput(self.X, self.V)

    #following funtion is to initialize the weights
    def findWeights(self,rows,column):
        if ( type(rows) is int and type(column) is int):
            W = np.empty((0, (column)))
            for i in range(rows):
                w = self.findPoints(-15, 15, column)
                W = np.vstack((W, w))
            return W
        else:
            print("Provide feasible values for input and output layers' nodes")

    def findPoints(self,a,b,n): #this function gets the random points
        x = list()
        for i in range(n):
            temp = random.uniform(a,b)
            x.append(temp)
        return x

    def getActualOutput(self,X,V): # this fucntion will calculate the required output
        D= list()
        for x,v in zip(X,V):
            d = math.sin(20 * x) + 3 * x + v
            D.append(d)
        return D

    def Output(self,x,W,A): # this function will calculate the local field and output
        I= []  # induced local field
        Z = [] # output field
        x = np.array(np.insert(x,0,1)).reshape(1,-1)
        for idx,(a,w) in enumerate(zip(A,W)):
            if (idx is 0):
                u = np.dot(w,x.T)
            else:
```

```
                u = np.insert(u,0,1)
                u = np.dot(w,u.T)
            I.append(np.array(u))
            u = np.array(self.findActivation(a,u))
            Z.append(u)
        return (I,Z)

    def findUpdate(self,x,d,W,no_of_layers,A,rate): # this function will perform the back propogation
        L = no_of_layers
        I, Z = self.Output(x, W, A)
        Delta = 0
        for i in reversed(range(L)):
            if i is (L-1):
                Delta = np.multiply((d - Z[i]), self.findDerivativeActivation(A[i],I[i]))[0] #
            else:
                W_n = np.delete(W[i+1],0,1)
                Delta = np.multiply(np.dot(W_n.T, Delta) , self.findDerivativeActivation(A[i],I[i]))
            if i is 0:
                Z_n = np.insert(np.array([x]), 0, 1).reshape(1,-1)
                Delta = Delta.reshape(1,-1)
                W[i] = W[i] + (rate) * np.dot((Delta).T, (Z_n))
            else:
                Z_n = np.insert(Z[i - 1], 0, 1)
                W[i] = W[i] + (rate) * np.dot((Delta), (Z_n))
        return (W)

    def findActivation(self, a, X):# this function will return the activation value
        tanh = np.vectorize(lambda x:math.tanh(x))
        relu = np.vectorize(lambda x:x)
        step = np.vectorize(lambda x:1 if x>=0 else 0)
        sigmoid = np.vectorize(lambda x: (math.exp(x)/ (1 + math.exp(x))))
        if a is 'tanh':
            y = tanh(X)
        elif a is 'relu':
            y = relu(X)
        elif a is 'step':
            y = step(X)
        elif a is 'softmax':
            y = self.softmax(X)
        elif a is 'sigmoid':
            y = sigmoid(X)
        return y

    def softmax_grad(self,s):
        jacobian_m = np.diag(s)
        for i in range(len(jacobian_m)):
            for j in range(len(jacobian_m)):
                if i == j:
                    jacobian_m[i][j] = s[i] * (1 - s[i])
```

```
                    else:
                        jacobian_m[i][j] = -s[i] * s[j]
        return jacobian_m

    def findDerivativeActivation(self,a,Y):# this computes derivative activation
        der_tanh = np.vectorize(lambda x:(1-math.tanh(x)**2))
        der_relu  = np.vectorize(lambda x:1)
        sigmoid = np.vectorize(lambda x: (math.exp(x) / (1 + math.exp(x))))
        if a is 'tanh':
            q = der_tanh(Y)
        elif a is 'relu':
            q = der_relu(Y)
        elif a is 'softmax':
            q = self.softmax_grad(Y)
        elif a is 'sigmoid':
            q =  (sigmoid(Y) * (1-(sigmoid(Y)**2)))
        return q

    def softmax(self,X):
        return (np.exp(X)/np.sum(np.exp(X)))

    def findOutputVec(self,W,X,A): # this will calculate the output vector
        Y = []
        for x in X:
            _,y = self.Output(x,W,A)
            Y.append(y[1])
        return Y

    def error(self,D,Y): #This calculates the Mean Squared Error
        MSE = 0
        for d,y in zip(D,Y):
            MSE += (d - y)**2
        return MSE/len(D)

    def plot(self,X,D,Y_out): #This will plot the graphs.
        plt.scatter(X, D, color='b',marker='o')
        plt.scatter(X,Y_out,color ='r',marker='x')
        plt.xlabel('x cordinate')
        plt.ylabel('y cordinate')
        plt.show()

# Following is the main function
if __name__ == '__main__':
    obj = Net()
    D = obj.D
    X = obj.X
    W_final = []
    # propogating to next layers using forward propogation
    no_of_nodes_hidden = 24
```

```
        W1 = obj.findWeights(no_of_nodes_hidden, 2)
        W2 = obj.findWeights(1, (no_of_nodes_hidden + 1))  # Ouput layer has one neuron with bias and preceeding
layer has N inputs
        W_final.append(W1)
        W_final.append(W2)
        A = ['tanh','relu']
        MSE = [] # to store mean sqaured error
        Y = obj.findOutputVec(W_final, X, A)     #Calculating the required output
        e = 0.015
        epoch = 0
  #Following perform back propogation
        while True:
            for x,d in zip(X,D):
                W_final = obj.findUpdate(x,d,W_final,no_of_layers=2,A =A, rate=0.01)
            Y = obj.findOutputVec(W_final,X,A)
            mse = obj.error(D,Y)
            print("This is the mean Squared Error",mse," at epoch ", epoch)
            MSE.append(mse)
            epoch += 1
            if ((MSE[epoch - 1] <= e) or (epoch>100)):     #Stop if MSE is less than e
                break
        range_epoch = [i for i in range(epoch)] #plots a graph of MSE with respect to the number of epochs.
        plt.plot(range_epoch,MSE)
        plt.show()
        Y = obj.findOutputVec(W_final, X, A)
        obj.plot(X,D,Y)
        plt.show()
```

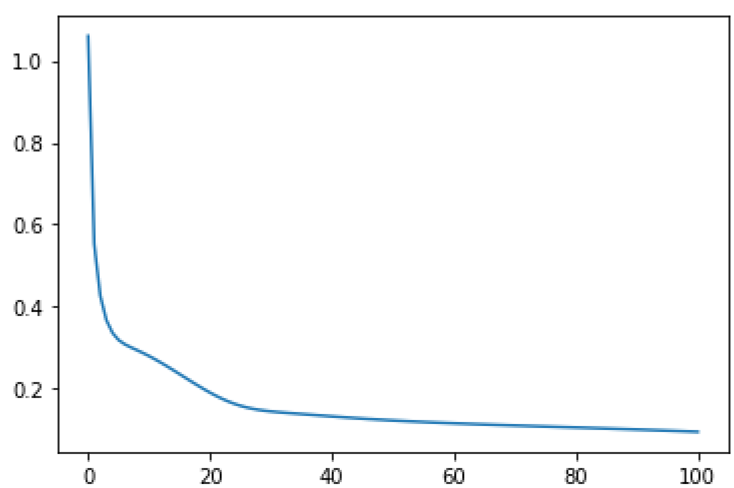
THE OUTPUT IS AS FOLLOWS:

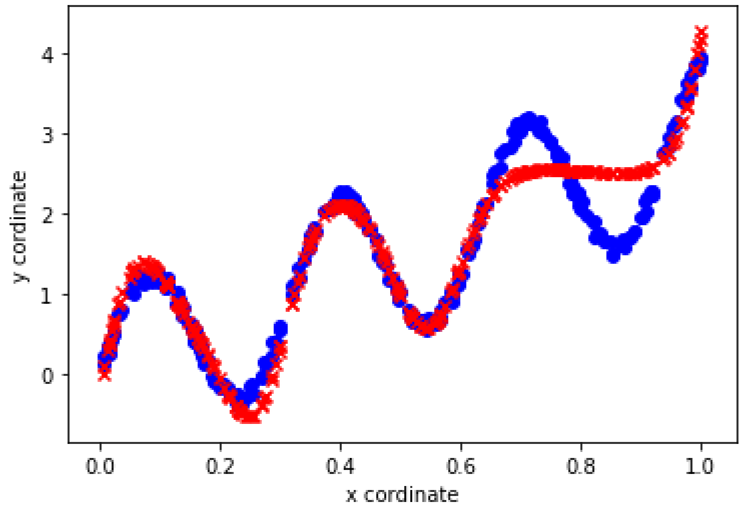
```
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.8.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/kaush/Downloads/NN/backPropagation.py',
wdir='C:/Users/kaush/Downloads/NN')
This is the mean Squared Error [1.06039443]  at epoch  0
This is the mean Squared Error [0.5523023]  at epoch  1
This is the mean Squared Error [0.42667395]  at epoch  2
This is the mean Squared Error [0.36749927]  at epoch  3
This is the mean Squared Error [0.33646227]  at epoch  4
This is the mean Squared Error [0.31909026]  at epoch  5
This is the mean Squared Error [0.30879679]  at epoch  6
This is the mean Squared Error [0.3012804]  at epoch  7
This is the mean Squared Error [0.29444015]  at epoch  8
This is the mean Squared Error [0.2874181]  at epoch  9
This is the mean Squared Error [0.2799192]  at epoch  10
This is the mean Squared Error [0.27189082]  at epoch  11
This is the mean Squared Error [0.2633827]  at epoch  12
This is the mean Squared Error [0.25448611]  at epoch  13
This is the mean Squared Error [0.24530766]  at epoch  14
This is the mean Squared Error [0.23595893]  at epoch  15
This is the mean Squared Error [0.22655325]  at epoch  16
This is the mean Squared Error [0.21720572]  at epoch  17
This is the mean Squared Error [0.2080343]  at epoch  18
This is the mean Squared Error [0.19916032]  at epoch  19
This is the mean Squared Error [0.19070742]  at epoch  20
This is the mean Squared Error [0.1827975]  at epoch  21
This is the mean Squared Error [0.17554348]  at epoch  22
This is the mean Squared Error [0.16903923]  at epoch  23
This is the mean Squared Error [0.1633481]  at epoch  24
This is the mean Squared Error [0.15849302]  at epoch  25
This is the mean Squared Error [0.15445096]  at epoch  26
This is the mean Squared Error [0.15115443]  at epoch  27
This is the mean Squared Error [0.14850039]  at epoch  28
This is the mean Squared Error [0.14636466]  at epoch  29
This is the mean Squared Error [0.14461845]  at epoch  30
This is the mean Squared Error [0.14314281]  at epoch  31
This is the mean Squared Error [0.14183851]  at epoch  32
This is the mean Squared Error [0.14063041]  at epoch  33
This is the mean Squared Error [0.13946712]  at epoch  34
This is the mean Squared Error [0.13831759]  at epoch  35
This is the mean Squared Error [0.13716646]  at epoch  36
This is the mean Squared Error [0.1360092]  at epoch  37
This is the mean Squared Error [0.13484801]  at epoch  38
This is the mean Squared Error [0.13368868]  at epoch  39
This is the mean Squared Error [0.13253833]  at epoch  40
This is the mean Squared Error [0.13140406]  at epoch  41
This is the mean Squared Error [0.13029211]  at epoch  42
This is the mean Squared Error [0.12920752]  at epoch  43
This is the mean Squared Error [0.12815402]  at epoch  44
This is the mean Squared Error [0.12713414]  at epoch  45
This is the mean Squared Error [0.12614929]  at epoch  46
This is the mean Squared Error [0.12520001]  at epoch  47
This is the mean Squared Error [0.12428609]  at epoch  48
This is the mean Squared Error [0.12340681]  at epoch  49
This is the mean Squared Error [0.12256101]  at epoch  50
This is the mean Squared Error [0.12174726]  at epoch  51
This is the mean Squared Error [0.12096394]  at epoch  52
This is the mean Squared Error [0.12020933]  at epoch  53
This is the mean Squared Error [0.11948163]  at epoch  54
This is the mean Squared Error [0.11877907]  at epoch  55
This is the mean Squared Error [0.11809988]  at epoch  56
```

```
This is the mean Squared Error [0.11744234]  at epoch  57
This is the mean Squared Error [0.11680478]  at epoch  58
This is the mean Squared Error [0.11618564]  at epoch  59
This is the mean Squared Error [0.11558338]  at epoch  60
This is the mean Squared Error [0.11499659]  at epoch  61
This is the mean Squared Error [0.11442391]  at epoch  62
This is the mean Squared Error [0.11386405]  at epoch  63
This is the mean Squared Error [0.11331582]  at epoch  64
This is the mean Squared Error [0.11277808]  at epoch  65
This is the mean Squared Error [0.11224977]  at epoch  66
This is the mean Squared Error [0.11172988]  at epoch  67
This is the mean Squared Error [0.11121745]  at epoch  68
This is the mean Squared Error [0.1107116]  at epoch  69
This is the mean Squared Error [0.11021149]  at epoch  70
This is the mean Squared Error [0.10971631]  at epoch  71
This is the mean Squared Error [0.1092253]  at epoch  72
This is the mean Squared Error [0.10873774]  at epoch  73
This is the mean Squared Error [0.10825296]  at epoch  74
This is the mean Squared Error [0.10777029]  at epoch  75
This is the mean Squared Error [0.1072891]  at epoch  76
This is the mean Squared Error [0.10680881]  at epoch  77
This is the mean Squared Error [0.10632882]  at epoch  78
This is the mean Squared Error [0.10584858]  at epoch  79
This is the mean Squared Error [0.10536754]  at epoch  80
This is the mean Squared Error [0.10488519]  at epoch  81
This is the mean Squared Error [0.104401]  at epoch  82
This is the mean Squared Error [0.10391448]  at epoch  83
This is the mean Squared Error [0.10342514]  at epoch  84
This is the mean Squared Error [0.10293248]  at epoch  85
This is the mean Squared Error [0.10243604]  at epoch  86
This is the mean Squared Error [0.10193533]  at epoch  87
This is the mean Squared Error [0.1014299]  at epoch  88
This is the mean Squared Error [0.10091928]  at epoch  89
This is the mean Squared Error [0.10040299]  at epoch  90
This is the mean Squared Error [0.09988057]  at epoch  91
This is the mean Squared Error [0.09935157]  at epoch  92
This is the mean Squared Error [0.09881551]  at epoch  93
This is the mean Squared Error [0.09827192]  at epoch  94
This is the mean Squared Error [0.09772034]  at epoch  95
This is the mean Squared Error [0.09716029]  at epoch  96
This is the mean Squared Error [0.09659129]  at epoch  97
This is the mean Squared Error [0.09601287]  at epoch  98
This is the mean Squared Error [0.09542455]  at epoch  99
This is the mean Squared Error [0.09482583]  at epoch  100
```





In [2]:

6. Pseudo Code

Following is for Training the model, all the update equations are written wherever necessary.

(i) Update equations for Forward Propogation:-

$$V = x * W_1 + b_1 \qquad (24 \times 1)$$
$$y = \tanh(V) \qquad (24 \times 1)$$
$$y_{pred} = \text{dot}(y, W_2) + b_2 \qquad (1 \times 1)$$

(ii) Update equations for backward propogation:-
(As specified in lecture)

$$\delta_{last} = 2(d - y_{pred})\,\phi'(V)$$
$$\delta_{last} = 2(d - y_{pred})$$
$$\delta_1 = \delta_{last} \cdot W_2 \cdot \tanh'(V)$$
$$E = (d_i - y_{pred})^2, \quad \frac{\partial E}{\partial V} = 2(d_i - y_{pred}); \quad \frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

In order to update the weights we perform the following

$$W = W + \eta \cdot \delta_1 \cdot x \qquad W_2 = W_2 + \eta \cdot \delta_{last} \cdot y$$
$$b = b + \eta\,\delta_1 \qquad b_2 = b_2 + \eta\,\delta_{last}$$

(iii) - Collect 300 points randomly from the uniform distribution [0,1].
- This is X.
- Compute V by getting 300 points from the uniform distribution [-1/10, 1/10] randomly.
- Compute d as follows
  $d = \sin(2X) + 3X + V$

(iv) Form a network that has 1 neuron in input layer, 24 in hidden layer and 1 in the output layer

(V) Finally, perform epochs until convergence or till no. of epochs is 100.
for each epoch(e):
For every sample in X:
perform forward propogation.
perform backward propogation.
Now you have the gradient.

Therefore, update the weights.

Compute Mean Squared error by taking average of the squared error for all data points.

if this value is greater than previous value of MSE
update learning rate ~~by~~ by multiply it with 0.9.
when MSE is less than 0.01. break.

(vii) End. (You can plot the graph to see how the algorithm converges).