```
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: