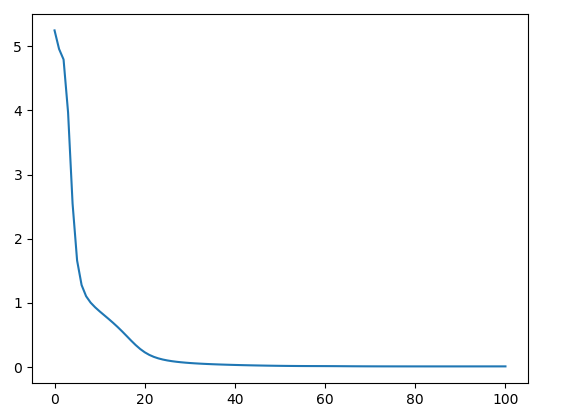
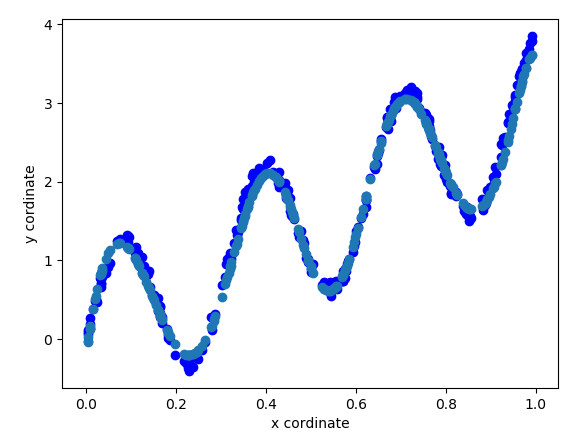
CS -559 – Neural Netwok , Fall -02018

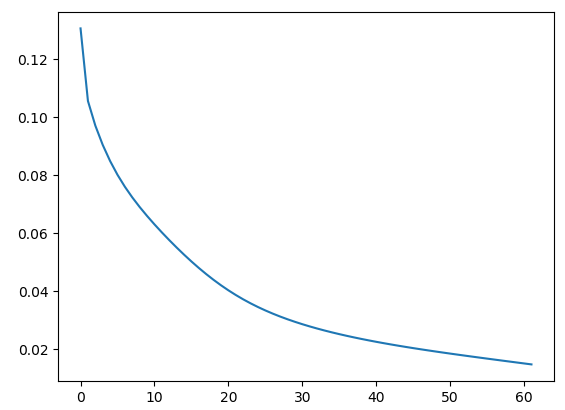
Q1) d) Plot of some of the trials on different set of initial weights



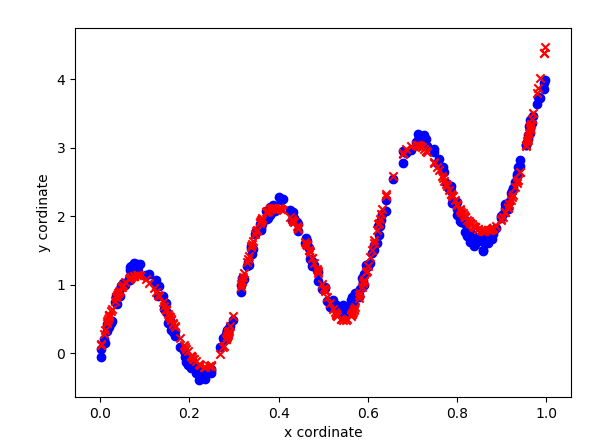
Error vs Epoch



Fit of the curve with the original



Error vs Epoch



Fit of the curve with the original

Source Code:

**import** numpy **as** np  
**import** random  
**import** math  
**import** matplotlib.pyplot **as** plt  
**from** random **import** shuffle  
  
  
N = 300 *# no of points in 2 layer  
  
# for a specific case for q1 ; will update for general case***class** Neural\_Network:  
 **def** \_\_init\_\_(self):  
 self.X = self.get\_random\_points(0, 1, N)  
 self.V = self.get\_random\_points(-1 / 10, 1 / 10, N)  
 self.D = self.get\_desired\_output(self.X, self.V)  
  
 *# initilaze the weights* **def** get\_weights(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.get\_random\_points(-15, 15, column)  
 W = np.vstack((W, w))  
 **return** W  
 **else**:  
 print(**"specify correct value for inpur and output layer nodes (int)"**)  
  
 *# Updated the previous function to randomly assign randomNeural\_Network  
 # points of any length with the specified range* **def** get\_random\_points(self,a,b,n):  
 x = list()  
 **for** i **in** range(n):  
 temp = random.uniform(a,b)  
 x.append(temp)  
 **return** x  
  
 **def** get\_desired\_output(self,X,V):  
 D= list()  
 **for** x,v **in** zip(X,V):  
 d = math.sin(20 \* x) + 3 \* x + v  
 D.append(d)  
 **return** D  
  
 *# forward pass  
 # getting the individual induced local field and output  
 # returns induced local field and outputs* **def** get\_Output(self,x,W,A):  
 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.get\_activation(a,u))  
 Z.append(u)  
 **return** (I,Z)  
  
 *# new function for update using equations* **def** get\_backpropagation\_update(self,x,d,W,no\_of\_layers,A,rate):  
 *# for l in range(no\_of\_layers):* L = no\_of\_layers  
 I, Z = self.get\_Output(x, W, A)  
 Delta = 0  
 **for** i **in** reversed(range(L)):  
 **if** i **is** (L-1):  
 Delta = np.multiply((d - Z[i]), self.get\_derivative\_activation(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.get\_derivative\_activation(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** softmax(self,X):  
 **return** (np.exp(X)/np.sum(np.exp(X)))  
  
 **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** get\_activation(self, a, X):  
 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** get\_derivative\_activation(self,a,Y):  
 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  
  
 *# @param X array , Y array* **def** graph(self,X,D,Y\_out):  
 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.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')* plt.show()  
  
 **def** calculate\_output\_vector(self,W,X,A):  
 Y = []  
 **for** x **in** X:  
 \_,y = self.get\_Output(x,W,A)  
 Y.append(y[1])  
 **return** Y  
  
 **def** get\_MSE(self,D,Y):  
 MSE = 0  
 **for** d,y **in** zip(D,Y):  
 MSE += (d - y)\*\*2  
 **return** MSE/len(D)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 ob = Neural\_Network()  
 D = ob.D  
 X = ob.X  
 X\_axis = [i **for** i **in** range(N)]  
 *# ob.graph(X,D, type = 'scatter') # @****TODO remove comment*** W\_final = []  
 *# fwd propagating to the next layers* no\_of\_nodes\_hidden = 24  
 W1 = ob.get\_weights(no\_of\_nodes\_hidden, 2) *# first layer has one input and we have* W2 = ob.get\_weights(1, (no\_of\_nodes\_hidden + 1)) *# output layer has one neuron,bias and previour layer with N inputs* W\_final.append(W1)  
 W\_final.append(W2)  
  
 A = [**'tanh'**,**'relu'**]  
  
 MSE = []  
 *### calculate desired output* Y = ob.calculate\_output\_vector(W\_final, X, A)  
  
 e = 0.015  
 epoch = 0  
 **while True**:  
 **for** x,d **in** zip(X,D):  
 W\_final = ob.get\_backpropagation\_update(x,d,W\_final,no\_of\_layers=2,A =A, rate=0.01)  
 Y = ob.calculate\_output\_vector(W\_final,X,A)  
 mse = ob.get\_MSE(D,Y)  
 print(mse)  
 MSE.append(mse)  
 epoch += 1  
 **if** ((MSE[epoch - 1] <= e) **or** (epoch>100)): *# break if value decreases below that value* **break** range\_epoch = [i **for** i **in** range(epoch)]  
 plt.plot(range\_epoch,MSE)  
 plt.show()  
  
 Y = ob.calculate\_output\_vector(W\_final, X, A)  
 ob.graph(X,D,Y)  
 plt.show()

Q2) Network Topology

Architecture Details

For the digit classification of MNIST dataset, I have used 2 layer neural network, 784 \* 24 \* 10. So the output vector represents a vector for the digits Eg ([1 0 0 ..]) for 0. Activation function is sigmoid activation function as represents the value within the range of 0-1 and in our case we require the output vector to be between 0-1. Furthermore, sigmoid gives a probalistic values for the input within (0 1), hence making it easier to classify digits. The no of nodes for the hidden layer has been based on the previous setup. I have also normalized the input vector as sigmoid function was giving math error with raw input as pixel value range from 0-255.

Design Process:

So initially I started with un-normalized data that led to math error in case of sigmoid function. Also, initially I took step function but that had zero gradient so the back propagation algorithm could not work in that scenario. So I adjusted the learning rate to 0.1 – 0.001 as initially the learning rate was too high that led to overflow of values and the it was converging.

Source Code:

**import** numpy **as** np  
**import** random  
**import** math  
**import** matplotlib.pyplot **as** plt  
**import** MNIST  
**import** backPropagation  
  
  
**class** image:  
 **def** \_\_init\_\_(self):  
 ob = MNIST.Mnist()  
 self.trainingImgs = ob.trainingImgs  
 self.trainingLabels = ob.trainingLabels  
  
 *# initilaze the weights* **def** get\_weights(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.get\_random\_points(-15, 15, column)  
 W = np.vstack((W, w))  
 **return** W  
 **else**:  
 print(**"specify correct value for inpur and output layer nodes (int)"**)  
  
 *# Updated the previous function to randomly assign randomNeural\_Network  
 # points of any length with the specified range* **def** get\_random\_points(self, a, b, n):  
 x = list()  
 **for** i **in** range(n):  
 temp = random.uniform(a, b)  
 x.append(temp)  
 **return** x  
  
 **def** calculate\_output\_vector(self,W,X,A):  
 Y = []  
 **for** x **in** X:  
 \_,y = self.get\_Output(x,W,A)  
 Y.append(y[1])  
 **return** Y  
  
 **def** encode\_labels(self):  
 labels = self.trainingLabels  
 D = [] *# desired output* d = np.array([])  
 **for** label **in** labels:  
 label = int(label)  
 **if** label == 0:  
 d = np.array([[1], [0], [0], [0], [0], [0], [0], [0], [0], [0]])  
 **elif** label **is** 1:  
 d = np.array([[0], [1], [0], [0], [0], [0], [0], [0], [0], [0]])  
 **elif** label == 2:  
 d = np.array([[0], [0], [1], [0], [0], [0], [0], [0], [0], [0]])  
 **elif** label == 3:  
 d = np.array([[0], [0], [0], [1], [0], [0], [0], [0], [0], [0]])  
 **elif** label == 4:  
 d = np.array([[0], [0], [0], [0], [1], [0], [0], [0], [0], [0]])  
 **elif** label == 5:  
 d = np.array([[0], [0], [0], [0], [0], [1], [0], [0], [0], [0]])  
 **elif** label == 6:  
 d = np.array([[0], [0], [0], [0], [0], [0], [1], [0], [0], [0]])  
 **elif** label == 7:  
 d = np.array([[0], [0], [0], [0], [0], [0], [0], [1], [0], [0]])  
 **elif** label == 8:  
 d = np.array([[0], [0], [0], [0], [0], [0], [0], [0], [1], [0]])  
 **elif** label == 9:  
 d = np.array([[0], [0], [0], [0], [0], [0], [0], [0], [0], [1]])  
 D.append(d)  
 **return** D  
  
 *# forward pass  
 # getting the individual induced local field and output  
 # returns induced local field and outputs* **def** get\_Output(self, x, W, A):  
 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.matrix(self.get\_activation(a, u)) *# to change it to 2-D* Z.append(u)  
 **return** (I, Z)  
  
 *# new function for update using equations* **def** get\_backpropagation\_update(self, x, d, W, no\_of\_layers, A, rate):  
 *# for l in range(no\_of\_layers):* L = no\_of\_layers  
 I, Z = self.get\_Output(x, W, A)  
 **for** i **in** reversed(range(L)):  
 **if** i **is** (L - 1):  
 Delta = np.multiply((d - Z[i]), self.get\_derivative\_activation(A[i], I[i])) *#* **else**:  
 W\_n = np.delete(W[i + 1], 0, 1)  
 Delta = np.multiply(np.dot(W\_n.T, Delta), self.get\_derivative\_activation(A[i], I[i]))  
 **if** i **is** 0:  
 Z\_n = np.insert(np.array([x]), 0, 1).reshape(1, -1)  
 W[i] = W[i] + (rate) \* np.dot((Delta), (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** get\_activation(self, a, X):  
 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** get\_derivative\_activation(self,a,Y):  
 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** normalize\_input(self,X):  
 normalize = np.vectorize(**lambda** x:x/255)  
 **return** (normalize(X))  
  
 **def** sigmoid(self,X):  
 **return** (np.exp(X)/ (1+ np.exp(X)))  
  
 **def** softmax(self,X):  
 **return** (np.exp(X)/np.sum(np.exp(X)))  
  
 **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** graph(self,X,D,Y\_out):  
 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.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')* plt.show()  
  
 *# @param self , output vector* **def** get\_mse\_mnist(self,Y):  
 error = 0  
 labels = self.trainingLabels  
 **for** label,y **in** zip(labels,Y):  
 j = np.argmax(y,axis=1)  
 **if** label != j[0]:  
 error += 1  
 **return** (error/len(Y))  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 ob = image()  
 ob1 = backPropagation.Neural\_Network()  
 images = ob.trainingImgs  
 labels = ob.trainingLabels  
  
 W\_final = []  
 N = 24 *# hidden layer* D = ob.encode\_labels()  
 images\_n = []  
 **for** image **in** images:  
 image = ob.normalize\_input(image)  
 images\_n.append(image)  
 W1 = ob1.get\_weights(N,(784+1))  
 W2 = ob1.get\_weights(10,(N+1))  
 W\_final.append(W1)  
 W\_final.append(W2)  
  
 e = 0.015  
 epoch = 0  
 A = [**'sigmoid'**,**'sigmoid'**]  
 MSE = []  
 **while True**:  
 **for** idx,(x,d) **in** enumerate(zip(images\_n,D)):  
 W\_final = ob.get\_backpropagation\_update(x, d, W\_final, no\_of\_layers=2, A=[**'sigmoid'**,**'sigmoid'**], rate=0.0009)  
  
 Y = ob.calculate\_output\_vector(W\_final, images\_n, A=[**'sigmoid'**,**'sigmoid'**])  
 mse = ob.get\_mse\_mnist(Y)  
 epoch = epoch + 1  
 print(mse)  
 print(epoch)  
 MSE.append(mse)  
 **if** ((MSE[epoch - 1] <= e) **or** (epoch>100)): *# break if value decreases below that value* **break** *# if ((MSE[epoch - 1] <= e) or (epoch > 100)): # break if value decreases below that value  
 # break*