Practical 4 Feed Forward Back-Propogation

import numpy as np  
  
#sigmoid activation function  
def sigmoid(x):  
 return 1 / (1+np.exp(-x))

import numpy as np  
  
# Load the training data  
training\_data = np.loadtxt('mnist\_train.csv', delimiter=',', dtype=np.float32, skiprows=1)  
  
# Load the test data  
test\_data = np.loadtxt('mnist\_test.csv', delimiter=',', dtype=np.float32, skiprows=1)

print("training\_data.shape = ", training\_data.shape, " , test\_data.shape = ", test\_data.shape)

training\_data.shape = (60000, 785) , test\_data.shape = (10000, 785)

class NeuralNetwork:  
  
 def \_\_init\_\_(self, input\_nodes, hidden\_nodes, output\_nodes, learning\_rate):  
  
 self.input\_nodes = input\_nodes  
 self.hidden\_nodes = hidden\_nodes  
 self.output\_nodes = output\_nodes  
  
 # Weight Initialization with Xavier/He : W2  
 self.W2 = np.random.randn(self.input\_nodes, self.hidden\_nodes) / np.sqrt(self.input\_nodes/2)  
 self.b2 = np.random.rand(self.hidden\_nodes)  
  
 # Weight Initialization Xavier/He : W3  
 self.W3 = np.random.randn(self.hidden\_nodes, self.output\_nodes) / np.sqrt(self.hidden\_nodes/2)  
 self.b3 = np.random.rand(self.output\_nodes)  
  
 # Initialization A3,Z3 : A3 is the result of sigmoid function about Z2  
 self.Z3 = np.zeros([1,output\_nodes])  
 self.A3 = np.zeros([1,output\_nodes])  
  
 # Initialization A2,Z2  
 self.Z2 = np.zeros([1,hidden\_nodes])  
 self.A2 = np.zeros([1,hidden\_nodes])  
  
 # Initialization A1,Z1  
 self.Z1 = np.zeros([1,input\_nodes])  
 self.A1 = np.zeros([1,input\_nodes])  
  
 # Learning rate Initialization  
 self.learning\_rate = learning\_rate  
  
 def feed\_forward(self):  
  
 delta = 1e-7 # log Infinite Divergence Prevention  
  
 # Calculate Z1,A1 in the input layer  
 self.Z1 = self.input\_data  
 self.A1 = self.input\_data  
  
 # Calculate Z2,A2 in the hidden layer  
 self.Z2 = np.dot(self.A1, self.W2) + self.b2  
 self.A2 = sigmoid(self.Z2)  
  
 # Calculate Z3,A3 in the ouput layer  
 self.Z3 = np.dot(self.A2, self.W3) + self.b3  
 self.A3 = sigmoid(self.Z3)  
  
 # Calculate the loss function value (error) : cross entropy  
 return -np.sum( self.target\_data\*np.log(self.A3 + delta) + (1-self.target\_data)\*np.log((1 - self.A3)+delta ) )  
  
 # For external printing  
 def loss\_val(self):  
  
 delta = 1e-7 # log Infinite Divergence Prevention  
  
 # Calculate Z1,A1 in the input layer  
 self.Z1 = self.input\_data  
 self.A1 = self.input\_data  
  
 # Calculate Z2,A2 in the hidden layer  
 self.Z2 = np.dot(self.A1, self.W2) + self.b2  
 self.A2 = sigmoid(self.Z2)  
  
 # Calculate Z3,A3 in the ouput layer  
 self.Z3 = np.dot(self.A2, self.W3) + self.b3  
 self.A3 = sigmoid(self.Z3)  
  
 # Calculate the loss function value : cross entropy  
 return -np.sum( self.target\_data\*np.log(self.A3 + delta) + (1-self.target\_data)\*np.log((1 - self.A3)+delta ) )  
  
 def train(self, input\_data, target\_data): # input\_data : 784 , target\_data : 10  
  
 self.target\_data = target\_data  
 self.input\_data = input\_data  
  
 # Calculate an error with the feed foward  
 loss\_val = self.feed\_forward()  
  
 # Calculate loss\_3  
 loss\_3 = (self.A3-self.target\_data) \* self.A3 \* (1-self.A3)  
  
 # Update W3, b3  
 self.W3 = self.W3 - self.learning\_rate \* np.dot(self.A2.T, loss\_3)  
  
 self.b3 = self.b3 - self.learning\_rate \* loss\_3  
  
 # Caculate loss\_2  
 loss\_2 = np.dot(loss\_3, self.W3.T) \* self.A2 \* (1-self.A2)  
  
 # Update W2, b2  
 self.W2 = self.W2 - self.learning\_rate \* np.dot(self.A1.T, loss\_2)  
  
 self.b2 = self.b2 - self.learning\_rate \* loss\_2  
  
 def predict(self, input\_data): # Shape of input\_data is (1, 784) matrix  
  
 Z2 = np.dot(input\_data, self.W2) + self.b2  
 A2 = sigmoid(Z2)  
  
 Z3 = np.dot(A2, self.W3) + self.b3  
 A3 = sigmoid(Z3)  
  
 predicted\_num = np.argmax(A3)  
  
 return predicted\_num  
  
 # Accuracy measurement  
 def accuracy(self, test\_data):  
  
 matched\_list = []  
 not\_matched\_list = []  
  
 for index in range(len(test\_data)):  
  
 label = int(test\_data[index, 0])  
  
 # Data normalize for one-hot encoding  
 data = (test\_data[index, 1:] / 255.0 \* 0.99) + 0.01  
  
  
 # Vector -> Matrix (for the prediction)  
 predicted\_num = self.predict(np.array(data, ndmin=2))  
  
 if label == predicted\_num:  
 matched\_list.append(index)  
 else:  
 not\_matched\_list.append(index)  
  
 print("Current Accuracy = ", 100\*(len(matched\_list)/(len(test\_data))), " %")  
  
 return matched\_list, not\_matched\_list

# Define variables  
input\_nodes = 784  
hidden\_nodes = 100  
output\_nodes = 10  
learning\_rate = 0.3  
epochs = 1  
  
nn = NeuralNetwork(input\_nodes, hidden\_nodes, output\_nodes, learning\_rate)  
  
for i in range(epochs):  
  
 for step in range(len(training\_data)): # train  
  
 # input\_data, target\_data normalize  
 target\_data = np.zeros(output\_nodes) + 0.01  
 target\_data[int(training\_data[step, 0])] = 0.99  
 input\_data = ((training\_data[step, 1:] / 255.0) \* 0.99) + 0.01  
  
 nn.train( np.array(input\_data, ndmin=2), np.array(target\_data, ndmin=2) )  
  
  
 # Print the error once every 400 times  
 if step % 400 == 0:  
 print("step = ", step, ", loss\_val = ", nn.loss\_val())

step = 0 , loss\_val = 5.181760451497374  
step = 400 , loss\_val = 1.8257279028072664  
step = 800 , loss\_val = 1.1966792019184942  
step = 1200 , loss\_val = 0.6891300029086208  
step = 1600 , loss\_val = 1.3263726552853805  
step = 2000 , loss\_val = 1.3204952536331338  
step = 2400 , loss\_val = 0.7099982590623498  
step = 2800 , loss\_val = 0.8521348818927709  
step = 3200 , loss\_val = 0.7284323585177378  
step = 3600 , loss\_val = 0.6745150319724835  
step = 4000 , loss\_val = 0.9315049798047481  
step = 4400 , loss\_val = 0.8098864914446494  
step = 4800 , loss\_val = 0.9004415456447329  
step = 5200 , loss\_val = 0.7883906736536096  
step = 5600 , loss\_val = 1.4187366952122564  
step = 6000 , loss\_val = 0.8133478931051695  
step = 6400 , loss\_val = 0.8920518709408244  
step = 6800 , loss\_val = 0.8922103035081866  
step = 7200 , loss\_val = 0.7958953427236826  
step = 7600 , loss\_val = 0.8275873918472992  
step = 8000 , loss\_val = 0.9278295180004442  
step = 8400 , loss\_val = 0.8049473009988288  
step = 8800 , loss\_val = 0.8621537099864216  
step = 9200 , loss\_val = 0.8422327118291508  
step = 9600 , loss\_val = 0.9464298623647481  
step = 10000 , loss\_val = 0.9536492573998673  
step = 10400 , loss\_val = 0.8734753626980161  
step = 10800 , loss\_val = 3.675958583100478  
step = 11200 , loss\_val = 0.8819966092171273  
step = 11600 , loss\_val = 12.732137584474202  
step = 12000 , loss\_val = 0.8663142245921063  
step = 12400 , loss\_val = 0.8288989471983186  
step = 12800 , loss\_val = 0.8958811066026711  
step = 13200 , loss\_val = 0.9477514256059684  
step = 13600 , loss\_val = 0.9771052074340931  
step = 14000 , loss\_val = 0.9139168996559212  
step = 14400 , loss\_val = 0.9797040212717119  
step = 14800 , loss\_val = 0.9134351486248249  
step = 15200 , loss\_val = 1.0273117262179081  
step = 15600 , loss\_val = 0.9459549613237408  
step = 16000 , loss\_val = 0.9370502930116031  
step = 16400 , loss\_val = 0.926825081423224  
step = 16800 , loss\_val = 1.0222358769723396  
step = 17200 , loss\_val = 0.9913980729988706  
step = 17600 , loss\_val = 0.9948514353196204  
step = 18000 , loss\_val = 0.9784320273347243  
step = 18400 , loss\_val = 0.8646093671727421  
step = 18800 , loss\_val = 0.970709409188004  
step = 19200 , loss\_val = 0.9789854462811967  
step = 19600 , loss\_val = 0.8880272323767441  
step = 20000 , loss\_val = 1.014880687723039  
step = 20400 , loss\_val = 0.9932301521358762  
step = 20800 , loss\_val = 1.0366683613197905  
step = 21200 , loss\_val = 0.888243954561937  
step = 21600 , loss\_val = 0.9828409396292421  
step = 22000 , loss\_val = 1.0292027330203348  
step = 22400 , loss\_val = 0.9400829594412163  
step = 22800 , loss\_val = 1.0095732978668155  
step = 23200 , loss\_val = 0.952914176908396  
step = 23600 , loss\_val = 1.0963917524746838  
step = 24000 , loss\_val = 0.998082233783454  
step = 24400 , loss\_val = 1.0072609542737156  
step = 24800 , loss\_val = 0.935318092182861  
step = 25200 , loss\_val = 1.0260818025345526  
step = 25600 , loss\_val = 0.9511676724756885  
step = 26000 , loss\_val = 0.9329197843977816  
step = 26400 , loss\_val = 1.0202691486539932  
step = 26800 , loss\_val = 1.1925423465374532  
step = 27200 , loss\_val = 1.0635290267817847  
step = 27600 , loss\_val = 1.1495830596243815  
step = 28000 , loss\_val = 1.0134522653551896  
step = 28400 , loss\_val = 1.1308291417563583  
step = 28800 , loss\_val = 0.9676096841575401  
step = 29200 , loss\_val = 1.0713780458262199  
step = 29600 , loss\_val = 1.0348181186531074  
step = 30000 , loss\_val = 1.094120696942325  
step = 30400 , loss\_val = 0.9708737338115005  
step = 30800 , loss\_val = 1.051621537924402  
step = 31200 , loss\_val = 9.504856045568117  
step = 31600 , loss\_val = 5.007270664603412  
step = 32000 , loss\_val = 0.9999164319619791  
step = 32400 , loss\_val = 1.0159900458145372  
step = 32800 , loss\_val = 1.0415436140478367  
step = 33200 , loss\_val = 1.0587872530357552  
step = 33600 , loss\_val = 0.960934242224828  
step = 34000 , loss\_val = 1.0684291060709432  
step = 34400 , loss\_val = 1.1160095233709213  
step = 34800 , loss\_val = 4.490555610971476  
step = 35200 , loss\_val = 1.1706698742528365  
step = 35600 , loss\_val = 0.9325540858000488  
step = 36000 , loss\_val = 0.9170898809988741  
step = 36400 , loss\_val = 1.1091688710263083  
step = 36800 , loss\_val = 1.039886302908426  
step = 37200 , loss\_val = 0.989936497979745  
step = 37600 , loss\_val = 1.0573129205280531  
step = 38000 , loss\_val = 0.9888541111990137  
step = 38400 , loss\_val = 1.1376954957792667  
step = 38800 , loss\_val = 1.02916776270281  
step = 39200 , loss\_val = 1.131361816829566  
step = 39600 , loss\_val = 4.196037192135921  
step = 40000 , loss\_val = 1.0349109569879413  
step = 40400 , loss\_val = 1.2026827478544782  
step = 40800 , loss\_val = 0.9791194230614335  
step = 41200 , loss\_val = 1.0144174839018245  
step = 41600 , loss\_val = 1.016301440766546  
step = 42000 , loss\_val = 0.9688043160664901  
step = 42400 , loss\_val = 1.0354544781605755  
step = 42800 , loss\_val = 1.0022507062196544  
step = 43200 , loss\_val = 1.2282255345834636  
step = 43600 , loss\_val = 0.9842832452718742  
step = 44000 , loss\_val = 1.1034690364199824  
step = 44400 , loss\_val = 1.054508434246422  
step = 44800 , loss\_val = 0.9767364205689235  
step = 45200 , loss\_val = 1.1279004359105729  
step = 45600 , loss\_val = 0.9404896391446581  
step = 46000 , loss\_val = 1.0413097098181316  
step = 46400 , loss\_val = 1.0853868171159233  
step = 46800 , loss\_val = 1.0671745399811905  
step = 47200 , loss\_val = 1.059654256341278  
step = 47600 , loss\_val = 1.0233522465569613  
step = 48000 , loss\_val = 1.0925473287579452  
step = 48400 , loss\_val = 1.1516931883402484  
step = 48800 , loss\_val = 1.050034534849957  
step = 49200 , loss\_val = 1.0971919004449338  
step = 49600 , loss\_val = 1.1020596653301826  
step = 50000 , loss\_val = 1.0763821246109675  
step = 50400 , loss\_val = 0.9720065753353822  
step = 50800 , loss\_val = 1.1232625989983602  
step = 51200 , loss\_val = 3.4229374503657057  
step = 51600 , loss\_val = 4.163693196381572  
step = 52000 , loss\_val = 1.0715220194961785  
step = 52400 , loss\_val = 1.1531190930894208  
step = 52800 , loss\_val = 8.902256155012715  
step = 53200 , loss\_val = 1.0980714264860163  
step = 53600 , loss\_val = 1.0840403543102572  
step = 54000 , loss\_val = 1.1432579756129464  
step = 54400 , loss\_val = 1.1377055077680087  
step = 54800 , loss\_val = 1.1484623338790905  
step = 55200 , loss\_val = 1.082493002547331  
step = 55600 , loss\_val = 1.0455505730728571  
step = 56000 , loss\_val = 0.950580047730449  
step = 56400 , loss\_val = 1.1090067785209603  
step = 56800 , loss\_val = 15.903588308117316  
step = 57200 , loss\_val = 1.0453790719820935  
step = 57600 , loss\_val = 1.1836292454267743  
step = 58000 , loss\_val = 1.1492854254840472  
step = 58400 , loss\_val = 1.0671682043018034  
step = 58800 , loss\_val = 1.1552433107855427  
step = 59200 , loss\_val = 1.0200300634945647  
step = 59600 , loss\_val = 1.2046299180874813

nn.accuracy(test\_data)

Current Accuracy = 93.84 %

([0,  
 1,  
 2,  
 3,  
 4,  
 5,  
 6,  
 7,  
 9,  
 10,  
 11,  
 12,  
 13,  
 14,  
 15,  
 16,  
 17,  
 18,  
 19,  
 20,  
 21,  
 22,  
 23,  
 24,  
 25,  
 26,  
 27,  
 28,  
 29,  
 30,  
 31,  
 32,  
 34,  
 35,  
 36,  
 37,  
 38,  
 39,  
 40,  
 41,  
 42,  
 43,  
 44,  
 45,  
 46,  
 47,  
 48,  
 49,  
 50,  
 51,  
 52,  
 53,  
 54,

...  
 9808,  
 9811,  
 9839,  
 9858,  
 9867,  
 9883,  
 9893,  
 9905,  
 9925,  
 9941,  
 9944,  
 9970,  
 9980,  
 9982])