**Artificial Intelligence Course**

**Assignment 07**



**Dr.Hashim Yasin**

Student Name: Abdul Salam & Taha Raja

Roll No: 18F-0326 & 18F-0137 Section: CS(6B)

Date performed: 02/07/2021

# Question no 01:

import numpy  
import array  
import math  
import numpy as np  
from pyexpat import model  
import pandas as pd  
from pandas import read\_csv  
from sklearn.model\_selection import train\_test\_split  
  
  
class NeuralNetwork:  
  
 def \_\_init\_\_(self, layersS, alpha=0.1):  
 self.W = []  
 self.layers = layersS  
 self.alpha = alpha  
 for i in np.arange(0, len(layersS) - 2):  
 w = np.random.randn(layersS[i] + 1, layersS[i + 1] + 1)  
 self.W.append(w / np.sqrt(layersS[i]))  
 # the last two layers are a special case where the input  
 # connections need a bias term but the output does not  
 w = np.random.randn(layersS[-2] + 1, layersS[-1])  
 self.W.append(w / np.sqrt(layersS[-2]))  
  
 def \_\_repr\_\_(self):  
 # construct and return a string that represents the network  
 # architecture  
 return "NeuralNetwork: {}".format("-".join(str(l) for l in self.layers))  
  
 def sigmoid(self, x):  
 return 1.0 / (1 + np.exp(-x))  
  
 def sigmoid\_deriv(self, x):  
 return x \* (1 - x)  
  
 def fit(self, X, y, epochs=1000, displayUpdate=100):  
 # insert a column of 1's as the last entry in the feature  
 # matrix -- this little trick allows us to treat the bias  
 # as a trainable parameter within the weight matrix  
 X = np.c\_[X, np.ones((X.shape[0]))]  
 # loop over the desired number of epochs  
 for epoch in np.arange(0, epochs):  
 # loop over each individual data point and train  
 # our network on it  
 for (x, target) in zip(X, y):  
 self.fit\_partial(x, target)  
 # check to see if we should display a training update  
 if epoch == 0 or (epoch + 1) % displayUpdate == 0:  
 loss = self.calculate\_loss(X, y)  
 print("[INFO] epoch={}, loss={:.7f}".format(epoch + 1, loss))  
  
 def fit\_partial(self, x, y):  
 # construct our list of output activations for each layer  
 # as our data point flows through the network; the first  
 # activation is a special case -- it's just the input  
 # feature vector itself  
 A = [np.atleast\_2d(x)]  
  
 # FEEDFORWARD:  
 # loop over the layers in the network  
 for layer in np.arange(0, len(self.W)):  
 # feedforward the activation at the current layer by  
 # taking the dot product between the activation and  
 # the weight matrix -- this is called the "net input"  
 # to the current layer  
 net = float(A[layer])\* float(self.W[layer])  
 # computing the "net output" is simply applying our  
 # nonlinear activation function to the net input  
 out = self.sigmoid(net)  
 # once we have the net output, add it to our list of  
 # activations  
 A.append(out)  
 # BACKPROPAGATION  
 # the first phase of backpropagation is to compute the  
 # difference between our \*prediction\* (the final output  
 # activation in the activations list) and the true target  
 # value  
 error = A[-1] - y  
 # from here, we need to apply the chain rule and build our  
 # list of deltas 'D'; the first entry in the deltas is  
 # simply the error of the output layer times the derivative  
 # of our activation function for the output value  
 D = [error \* self.sigmoid\_deriv(A[-1])]  
  
 # once you understand the chain rule it becomes super easy  
 # to implement with a 'for' loop -- simply loop over the  
 # layers in reverse order (ignoring the last two since we  
 # already have taken them into account)  
 for layer in np.arange(len(A) - 2, 0, -1):  
 # the delta for the current layer is equal to the delta  
 # of the \*previous layer\* dotted with the weight matrix  
 # of the current layer, followed by multiplying the delta  
 # by the derivative of the nonlinear activation function  
 # for the activations of the current layer  
 delta = D[-1].dot(self.W[layer].T)  
 delta = delta \* self.sigmoid\_deriv(A[layer])  
 D.append(delta)  
  
 # since we looped over our layers in reverse order we need to  
 # reverse the deltas  
 D = D[::-1]  
 # WEIGHT UPDATE PHASE  
 # loop over the layers  
 for layer in np.arange(0, len(self.W)):  
 # update our weights by taking the dot product of the layer  
 # activations with their respective deltas, then multiplying  
 # this value by some small learning rate and adding to our  
 # weight matrix -- this is where the actual "learning" takes  
 # place  
 self.W[layer] += -self.alpha \* A[layer].T.dot(D[layer])  
  
 def predict(self, X, addBias=True):  
 # initialize the output prediction as the input features -- this  
 # value will be (forward) propagated through the network to  
 # obtain the final prediction  
 p = np.atleast\_2d(X)  
 # check to see if the bias column should be added  
 if addBias:  
 # insert a column of 1's as the last entry in the feature  
 # matrix (bias)  
 p = np.c\_[p, np.ones((p.shape[0]))]  
 # loop over our layers in the network  
 for layer in np.arange(0, len(self.W)):  
 # computing the output prediction is as simple as taking  
 # the dot product between the current activation value 'p'  
 # and the weight matrix associated with the current layer,  
 # then passing this value through a nonlinear activation  
 # function  
 p = self.sigmoid(np.dot(p, self.W[layer]))  
 # return the predicted value  
 return p  
  
 def calculate\_loss(self, X, targets):  
 # make predictions for the input data points then compute  
 # the loss  
 targets = np.atleast\_2d(targets)  
 predictions = self.predict(X, addBias=False)  
 loss = 0.5 \* np.sum((predictions - targets) \*\* 2)  
 # return the loss  
 return loss  
  
class deltaRule:  
 def \_\_init\_\_(self):  
 self.Layers = 1  
 self.neuro = np.array([1])  
  
 def setLayer(self, layer):  
 self.Layers = int(layer)  
 newNeuro = []  
 for x in range(layer):  
 if (len(self.neuro) > x):  
 newNeuro.append(self.neuro[x])  
 else:  
 newNeuro.append([1])  
 self.neuro = newNeuro  
  
 def setNeurons(self, neurons, layer):  
 self.neuro[int(layer) - 1] = [int(neurons)]  
  
 def sigmo(self, y):  
 #sig = math.tanh(y)  
 sig = 1 / (1 + np.exp(-y))  
 return sig  
  
 def neuronWork(self, tranning\_x, weight\_x, tranning\_y, weight\_y, thresh, weight\_thresh):  
 eq = tranning\_x \* weight\_x + tranning\_y \* weight\_y + ((-thresh) \* weight\_thresh)  
 sigma = self.sigmo(eq)  
 return sigma  
  
 def solve(self):  
  
 url = 'SpiralData\_Clean\_Spread.csv'  
 names = ['Chem1', 'Chem2', 'Clnum']  
 dataset = read\_csv(url, names=names)  
 array = dataset.values  
 X = array[1:, 0:2]  
 y = array[1:, 2]  
 for i in range(len(X)):  
 X[i][0] = float(X[i][0])  
 X[i][1] = float(X[i][1])  
  
 y = array[1:, 2]  
 for i in range(len(y)):  
 y[i] = float(y[i])  
  
 input = X #[[0, 0], [0, 1], [1, 0], [1, 1]]  
 output =y # [0, 1, 1, 0]  
 weight = [[[-0.63504703, 0.48041041, -0.77972384],[-1.27247349, 0.40944617, 1.08593283]],[[-0.15318875, -1.75043772, 0.4891079 ], [-0.15318875, -1.75043772, 0.4891079 ]], [[0.05, 0.05, 0.05]]]  
 thresh = [[1, 1], [1, 1], [1]]  
 eta\_n = 0.9  
  
 preditct = []  
 sig\_hf = []  
 epoch = 10  
 counter = 0  
 incorrect = 1  
  
 # farward propagation  
 for ep in range(0, epoch):  
 print("epoch:", epoch)  
 for inp in range(len(input)):  
 input\_output = [input[inp]]  
 output\_actual = output[inp]  
 for l in range(len(self.neuro)):  
 y1y2 = []  
 for n in range(self.neuro[l][0]):  
  
 #for (x , y) in zip(input, output):  
 out = self.neuronWork(input\_output[l][0], weight[l][n][0],input\_output[l][1] , weight[l][n][1], thresh[l][n], weight[l][n][2])  
 y1y2.append(out)  
 input\_output.append(y1y2)  
  
  
 print("input", input\_output[0][0],"," , input\_output[0][1], " actual:",output\_actual ," -> predict:", out," weight:", weight[2])  
  
 #backward propagation  
 delta\_counter = self.Layers  
 del\_o = input\_output[delta\_counter][0] \* (1 - input\_output[self.Layers][0]) \* (output\_actual - input\_output[self.Layers][0])  
 delta = []  
 delta.append([del\_o])  
 for l in reversed(range(len(self.neuro)-1)):  
 del\_h = []  
 for n in range(self.neuro[l][0]):  
 del\_h.append(input\_output[l+1][n] \* (1 - input\_output[l+1][n]) \* delta[0][0] \* weight[l+1][0][n])  
 delta.insert(0, del\_h)  
  
 #find weight  
  
 for w in range(len(weight)):  
 for n in range(self.neuro[w][0]):  
 weight[w][n][0] = eta\_n \* delta[w][n] \* input\_output[w][0]  
 weight[w][n][1] = eta\_n \* delta[w][n] \* input\_output[w][1]  
 weight[w][n][2] = eta\_n \* delta[w][n] \* thresh[w][n]  
 epoch +=1  
 print("--------------------------------------------")  
 """  
 while (counter < epoch and incorrect > 0):  
 incorrect = 0  
 print("Epoch:", counter)  
 for k in range(len(input)):  
 for j in range(self.neuro[0][0]):  
 out = self.neuronWork(input[k][0], weight[0][j][0], input[k][1], weight[0][j][1], thresh[0][j],  
 weight[0][j][2])  
 preditct.append(out)  
  
 out = self.neuronWork(preditct[0], weight[1][0][0], preditct[1], weight[1][0][1], thresh[1][0],  
 weight[1][0][2])  
 out = round(out, 4)  
 # for i in range(len(self.neuro[1])):  
  
 print(input[k][0], " xor ", input[k][1], " --> Actual ", output[k], " predict:", out, " =>weight:",  
 weight[1][0])  
 if (output[k] != out):  
 incorrect += 1  
 sig\_o = out \* (1 - out) \* (output[k] - out) # k  
  
 for x in range(len(self.neuro[0])):  
 sig\_h = preditct[x] \* (1 - preditct[x]) \* (sig\_o \* weight[1][0][x])  
 sig\_h = round(sig\_h, 4)  
 sig\_hf.append(sig\_h)  
  
 for j in range(len(self.neuro[0])):  
 weight[0][j][2] = round(weight[0][j][2] + (eta\_n \* sig\_hf[j] \* thresh[0][j]), 4)  
 weight[0][j][0] = round(weight[0][j][0] + (eta\_n \* sig\_hf[j] \* input[k][j]), 4)  
 weight[0][j][1] = round(weight[0][j][1] + (eta\_n \* sig\_hf[j] \* input[k][j]), 4)  
  
 weight[1][0][2] = round(weight[1][0][2] + (eta\_n \* sig\_o \* thresh[1][j]), 4)  
 weight[1][0][0] = round(weight[1][0][0] + (eta\_n \* sig\_o \* preditct[0]), 4)  
 weight[1][0][1] = round(weight[1][0][1] + (eta\_n \* sig\_o \* preditct[1]), 4)  
  
 counter += 1  
 print(" -------------------------------------------------")  
 """  
  
url = 'SpiralData\_Clean\_Spread.csv'  
names = ['Chem1', 'Chem2', 'Clnum']  
dataset = read\_csv(url, names=names)  
array = dataset.values  
X = array[1:,0:2]  
#print(y)  
  
obj = deltaRule()  
obj.setLayer(3)  
obj.setNeurons(2, 1)  
obj.setNeurons(2, 2)  
obj.setNeurons(1, 3)  
obj.solve()  
  
  
  
# construct the XOR dataset  
#X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
#y = np.array([[0], [1], [1], [0]])  
nn = NeuralNetwork([2,2,1], alpha=0.5)  
#nn.fit(X, y, epochs=10)  
  
"""  
# now that our network is trained, loop over the XOR data points  
for (x, target) in zip(X, y):  
 # make a prediction on the data point and display the result  
 # to our console  
 pred = nn.predict(x)[0][0]  
 step = 1 if pred > 0.5 else 0  
 print("[INFO] data={}, ground-truth={}, pred={:.4f}, step={}".format(x, target[0], pred, step))  
  
"""

# **Searches screenshot:**