**Connectionism Assignment**

Luke Killoran

13434418

**Python Code:**

import os

os.chdir('C:/Users/Luke/Documents/UCD/Connectionism/Assignment')

import numpy as np

import numpy.random as random

import math

import pandas as pd

from sklearn.neural\_network import MLPRegressor

from sklearn import metrics

import random

class MyMLP():

"""

MLP

"""

def \_\_init\_\_(self, NI = 2, NH = 2, NO = 1):

# defined at start

self.NI = NI

self.NH = NH

self.NO = NO

# no definition required

self.W1 = self.dW1 = np.zeros((self.NH,self.NI)) # extra row wont be used just set to 1

self.W2 = self.dW2 = np.zeros((self.NO,self.NH))

self.Z1 = self.H = np.zeros((self.NH,1))

self.Z2 = np.zeros((self.NO,1))

self.O = np.zeros((self.NO,1))

# definition provided later

self.I = np.zeros((self.NI,1))

self.delta1 = np.zeros((self.NH,1))

self.delta2 = np.zeros((self.NO,1))

self.learning\_rate = 0.1

self.bias1 = np.zeros((self.NH,1)) # bias for each neuron in hidden layer

self.bias2 = np.zeros((self.NO,1))

def randomise(self,lower\_bound = 0.001, upper\_bound=0.4):

for row in range(self.dW1.shape[0]):

for col in range(self.dW1.shape[1]):

self.dW1[row][col] = random.uniform(lower\_bound, upper\_bound)

for row in range(self.dW2.shape[0]):

for col in range(self.dW2.shape[1]):

self.dW2[row][col] = random.uniform(lower\_bound, upper\_bound)

for row in range(self.bias1.shape[0]):

self.bias1[row] = random.uniform(lower\_bound, upper\_bound)

for row in range(self.bias2.shape[0]):

self.bias2[row] = random.uniform(lower\_bound, upper\_bound)

def forward(self,example\_input):

self.I = np.asarray(example\_input) # hopefully NI x 1

self.I.shape = (self.NI,1)

self.Z1 = np.dot(self.W1,self.I) + self.bias1

for row in range(self.Z1.shape[0]):

self.H[row] = 1/(1+math.exp(-self.Z1[row]))

self.H.shape = (self.NH,1)

self.Z2 = np.dot(self.W2,self.H) + self.bias2

for row2 in range(self.Z2.shape[0]):

self.O[row2] = 1/(1+math.exp(-self.Z2[row2]))

def double\_backwards(self,example\_output,learning\_rate = 0.1):

self.learning\_rate = learning\_rate

y = np.asarray(example\_output)

y.shape = (self.NO,1)

self.delta2 = y - self.O

nearly\_delta1 = np.dot(self.W2.transpose(),self.delta2)

for row in range(nearly\_delta1[0].shape[0]):

self.delta1[row] = nearly\_delta1[row]\*self.H[row]\*(1-self.H[row])

self.dW2 = self.learning\_rate\*np.dot(self.delta2,self.H.transpose())

self.dW1 = self.learning\_rate\*np.dot(self.delta1,self.I.transpose())

error = 0.5\*sum(np.power(self.delta2,2))

return error

def update\_weights(self):

self.bias1 += self.learning\_rate\*self.delta1\*0.5

self.bias2 += self.learning\_rate\*self.delta2\*0.5

self.W1 += self.dW1

self.W2 += self.dW2

self.dW1 = np.zeros((self.NH,self.NI))

self.dW2 = np.zeros((self.NO,self.NH))

# Task1: XOR

examples = [[[0,0],0],[[0,1],1],[[1,0],1],[[1,1],0]]

NN = MyMLP(NI=2,NH=2,NO=1)

NN.randomise(lower\_bound = -1/math.sqrt(2+2), upper\_bound = 1/math.sqrt(2+2)) # lb = 0.001, ub = 0.4, lr = 1

max\_epochs = 500000

for e in range(max\_epochs):

error = 0

for ex in range(len(examples)):

NN.forward(examples[ex][0])

out = np.asarray(examples[ex][1])

error += NN.double\_backwards(out,learning\_rate = 0.5)[0]

NN.update\_weights()

if e%50000==0: print("Error at epoch",e,"is",error)

predictions = []

for ex in range(len(examples)):

NN.forward(examples[ex][0])

predictions += [np.ndarray.tolist(NN.O)[0][0]]

# should be [0,1,1,0]

print(predictions)

# Task 2: Sin()

vectors = np.zeros((50,5))

for row in range(vectors.shape[0]):

for col in range(4):

vectors[row][col] = random.uniform(-1,1)

vectors[row][4] = math.sin(vectors[row][0]-vectors[row][1]+vectors[row][2]-vectors[row][3])

NN2 = MyMLP(NI=4,NH=5,NO=1)

NN2.randomise(lower\_bound = -0.5, upper\_bound=0.5)

max\_epochs = 10000

for e in range(max\_epochs):

error = 0

for ex in range(40):

NN2.forward(vectors[ex][0:4])

error += NN2.double\_backwards(vectors[ex][4],learning\_rate = 0.02)[0]

NN2.update\_weights()

if e%1000 == 0: print("Error at epoch",e,"is",error)

predictions = np.zeros((50,1))

y = vectors[0:50,4]

y.shape = (50,1)

for ex in range(0,50):

NN2.forward(vectors[ex][0:4])

predictions[ex] = NN2.O[0][0]

error = 0.5\*sum(np.power(predictions[40:50] - y[40:50],2))

print(error)

# this shows how low the predictions go

max(predictions)

min(predictions)

# Exceptional Task: Letter Recognition

# read in file

with open('letter-recognition.data') as myfile:

dataset = myfile.read()

# split observations and remove final empty one

dataset = dataset.split("\n")

dataset = dataset[:20000]

y\_letter = []

x = []

# split and x and y features

for i in range(len(dataset)):

row = dataset[i].split(",")

y\_letter += row[0]

x.append(row[1:])

for entry in enumerate(x[i]):

x[i][entry[0]] = float(int(entry[1]))

# normalise all x features and allot each letter and number ranking instead

x = np.asarray(x)

for col in range(x.shape[1]):

maxx = max(x[:,col])

minn = min(x[:,col])

for row in range(x.shape[0]):

x[row,col] = (x[row,col]-minn)/(maxx-minn)

y\_int = np.zeros((len(y\_letter),1),dtype=np.int)

y = np.zeros((len(y\_letter),26),dtype=np.int)

for k in range(len(y\_letter)):

y\_int[k] = ord(y\_letter[k])-65

y[k,y\_int[k][0]] = 1

# need random testing so here’s order

randomlist = random.sample(range(20000),20000)

NN3 = MyMLP(NI=16,NH=10,NO=26)

NN3.randomise(lower\_bound = -0.15, upper\_bound=0.15)

max\_epochs = 100000

for e in range(max\_epochs):

error = 0

for ex in range(16000):

NN3.forward(x[randomlist[ex]])

error += NN3.double\_backwards(y[randomlist[ex]],learning\_rate = 0.4)[0]

NN3.update\_weights()

if e%100 == 0: print("Error at epoch",e,"is",error)

# prob predictions are the probability that letter with greatest weighting is correct according to other weights

predictions = np.zeros((20000,26))

letter\_predictions = []

prob\_predictions = []

for ex in range(0,20000):

NN3.forward(x[randomlist[ex]])

col\_of\_max = np.argmax(np.max(NN3.O, axis=1))

letter\_predictions += [chr(col\_of\_max+65)]

prob\_predictions += [NN3.O[col\_of\_max][0]/sum(NN3.O)[0]]

for col in range(0,26):

predictions = NN3.O[col][0]

error = 0.5\*sum(sum(np.power(predictions[16000:] - y[randomlist[16000:]],2)))

print(error/4000)

sorted\_letter\_predictions = [x for \_,x in sorted(zip(randomlist,letter\_predictions),\

key=lambda pair: pair[0])]

print("Classification Acc on Whole Dataset: ",\

metrics.accuracy\_score(y\_letter, sorted\_letter\_predictions)\*100, "%")

num\_train\_correct = 0

num\_test\_correct = 0

for obs in range(16000):

if letter\_predictions[obs] == y\_letter[randomlist[obs]]:

num\_train\_correct+=1

for obs in range(16000,20000):

if letter\_predictions[obs] == y\_letter[randomlist[obs]]:

num\_test\_correct+=1

print("Classification Acc on Train Set:",num\_train\_correct\*100/16000,"%")

print("Classification Acc on Test Set:",num\_test\_correct\*100/4000,"%")

**Output:**

Task 1 (XOR):

Errors:

Error at epoch 0 is 0.770541337201

Error at epoch 50000 is 9.33005593939e-05

Error at epoch 100000 is 2.3707512912e-05

Error at epoch 150000 is 1.06465478926e-05

Error at epoch 200000 is 6.03471302062e-06

Error at epoch 250000 is 3.88574424592e-06

Error at epoch 300000 is 2.71204882649e-06

Error at epoch 350000 is 2.00110415986e-06

Error at epoch 400000 is 1.53784610024e-06

Error at epoch 450000 is 1.21912985868e-06

Predictions:

[8.758710870851507e-06, 0.9997251285187905, 0.9991441596427896, 0.0010813835616933325]

# should be [0,1,1,0]

Task 2 (Sin):

Error on Training Set:

Error at epoch 0 is 10.6403597075

Error at epoch 1000 is 5.61617252317

Error at epoch 2000 is 5.54690592052

Error at epoch 3000 is 5.57666307396

Error at epoch 4000 is 5.62788196024

Error at epoch 5000 is 5.68087391374

Error at epoch 6000 is 5.72874576812

Error at epoch 7000 is 5.76991624225

Error at epoch 8000 is 5.80482612194

Error at epoch 9000 is 5.83448157043

or 0.146 for each observation in training set

and Error on test set: 1.84561075 in total or 0.185 for each observation in test set

Predictions:

All predictions were lower than 1.60847878e-38

Exceptional Task (Letter Recognition):

Errors:

Error at epoch 0 is 7793.911834

Error at epoch 100 is 6138.15072874

Error at epoch 200 is 5922.25099513

Error at epoch 300 is 5808.50456696

Error at epoch 400 is 5743.58498173

Error at epoch 500 is 5688.23967901

Error at epoch 600 is 5653.1358139

Error at epoch 700 is 5625.81050808

Error at epoch 800 is 5603.73520965

Error at epoch 900 is 5586.00762275

Error at epoch 1000 is 5568.38303992

or Error is 0.318 per letter in training set

Error is 0.336 per letter in test set

Classification Acc on Train Set: 52.725 %

Classification Acc on Test Set: 52.55 %

**Analysis:**

Task 1 (XOR):

I ran an MLP with 2 inputs, 2 hidden neurons in one layer, and 1 output. There were 3 biases in total, 2 for the 2 hidden neurons and 1 for the output neuron. The weights and biases were initialised as random values between -0.5 and 0.5. The learning rate was set at 0.5. Each neuron had a sigmoid activation function. The learning was done through backpropagation were the delta of the output was simply the target (1 or 0) minus the output (prediction). The delta of each of the hidden neurons was the delta of the output multiplied by the weight from that hidden neuron to the output and multiplied by the activation value and 1 minus the activation value of that hidden neuron (which is the derivative of the sigmoid activation function of the sum of the weighted input activation values to that neuron). The change in weight for each destination neuron in the hidden layer and output layer and was simply the sum of the corresponding delta value for that neuron multiplied by the learning rate and the activation values of the neurons that are inputs to that neuron.

The MLP performed well, reducing its total error on the 4 observations to below 0.0001 within 50000 runs. The predictions were extremely close to the actual target values. MInksy’s observation that with perceptrons (with no hidden layers) cannot “learn” the XOR function is redundant since adding an extra layer allows the neural network to adequately model the function. This test is clear evidence of that.

Task 2 (Sin):

This task was programmed in extremely similar fashion to that of the previous task, with 2 extra inputs and 3 more hidden neurons but the formulas were unchanged, except that the learning rate was much lower, at 0.002 now.

The neural network performed poorly. Its error improved in the first 1000 runs but waivered after that, sometimes even increasing. The errors per observation on the training and test sets were 0.146 and 0.185 (27% higher) respectively – about one million times larger than on the previous task. What is worse is that it did not learn in any sense of the word – it predicted approximately 0 for all values. This is why the test classification error was so similar to the training one. However, all of this was to be expected. With only one output the neural network converges to one of two outputs: 0 or 1. Thus, it is not realistic to expect the network to be able to model a function with a real output using only one output neuron. This explains why all the predictions for the test values were below .

Exceptional Task (Letter Recognition):

This task was slightly different to the previous two in that it had 26 outputs so the values for the hidden deltas depended on more than one weight. Other than that, the rest of the formulas remained the same. The learning rate was 0.4 and the weights and biases were randomly initialised between -0.15 and 0.15.

The neural network performed adequately but not well. Its error always shrank but by a decreasing rate at each epoch. The errors per observation on the training and test sets were 0.316 and 0.336 respectively. The classification accuracies in the training and test sets were 52.75% and 52.55% respectively. This is not an amazing feat but compares well with random guessing (3.85%) and OneR (3.945%)

Note the classification accuracy on the training and test sets both improved by 9% (to over 61%) when I used 20 hidden neurons instead of 10. Perhaps if I tried more combinations of number of hidden neurons, learning rate, bias and weight initialisations, the algorithm would be more accurate.