James Hurt

# Introduction to Computational Intelligence: Computer Project 1

## Technical Description

I chose to implement my project in Python3 on Ubuntu 16.04 LTS due to its large number of libraries and the relaxed syntax of the language. The libraries used:

* NumPy is a library specializing in complicated mathematical structures. NumPy was the vector and ndarray handler, which is mathematically equivalent to a matric. I also used it for finding v = wTx as well as storing the weights and biases throughout the program
* Pandas is a library that has a simple and elegant way to parse CSV files into a structure called a Dataframe which resembles an SQL table. I used this library to bring in initial values for w and b as well as the training data with labels from CSV files. Pandas also has a function that converts a dataframe to a NumPy ndarray, making it a very easy library to implement
* PPrint is a Python library that allows for stylized printing such as how many items per line. I used this library to make the results of the network training (weights and bias) more legible.
* ArgParse has functions for passing arguments to the python file via command line. I wanted the user to be able to pass in the dimensions of the network and then input filenames for a general-purpose program. Then I went ahead and built in -a and -b parameters to allow for the script to automatically load the correct files for parts A and B of the assignment.
* MatPlotLib. This library is used for building graphs and plotting points and curves. I used this library to build the graphs asked for in part A, such as error per epoch and the data itself.

## Algorithm Design

The program is a Python implementation of a Multilayer Perceptron using Backpropagation. The design of the algorithm is in the following steps:

1. Setup
   1. Choose initial weights and bias
   2. Load training data
   3. Choose activation function
   4. Choose constants: Alpha, Beta, and Termination Threshold
2. Present point pi to the network and obtain the output
3. Calculate error of pi
4. Backpropagate
   1. Calculate the momentum term using predefined momentum constant and the previous change in wji
   2. Calculate the delta for neuron j
      1. For an output neuron, multiply the derivative of the activation function at the induced local field by error of the neuron
      2. For a hidden neuron, multiply the derivative of the activation function at the induced local field by dot product of the weights from this neuron to all neurons in the next layer with the deltas of those neurons in the next layer
   3. Multiply the delta for neuron j with the predefined constant for learning rate as well the value of the source neuron for weight wji
   4. Add the value calculated in part c to the current value of wji and assign that value as the next wji
5. Continue to perform epochs until the average error energy is below .001

The flow of this algorithm through my program is:

1. Init() – parse the arguments passed in from the command line and create the w1, w2, b1, b2, train\_data, and label matrices for which to run
2. Run() – randomize the data, and continue to run epochs until the termination condition is met
3. Epoch() – present each training point to the network and update weights
4. Show\_to \_layer() – present the dataset to a given layer with given weights and bias
5. Backpropagate() – update all weights given the output, labels, and current and previous values of the weights

I also have many helper functions to keep logic in one place, such as the fi or fi\_prime function. After the network has converged, I also have functions like print\_results to actually show the results in a legible way as well as functions to present the graphs needed for part A.

## Algorithm Results

The algorithm was not changed for parts A and B. The same functions were utilized for both parts of the assignment

### Part A

After running 1 epoch, the values I obtained for w1, w2, b1, and b2 were:

|  |  |
| --- | --- |
| **B1** | Bias |
| b1 | -0.1495 |
| b2 | 0.9248 |
| b3 | 0.4322 |
| b4 | -0.1462 |
| b5 | -0.7105 |
| b6 | -0.5103 |
| b7 | -0.705 |
| b8 | -0.9313 |
| b9 | -0.4693 |
| b10 | 1.1784 |

|  |  |  |
| --- | --- | --- |
| **W1** | From x1 | From x2 |
| w1 | 0.4292 | -1.1591 |
| w2 | 0.4619 | 0.3795 |
| w3 | 0.6854 | -0.2006 |
| w4 | 0.0679 | -0.2199 |
| w5 | 0.1865 | 0.1026 |
| w6 | -0.0906 | 0.9862 |
| w7 | 0.794 | -0.6615 |
| w8 | -0.6719 | 0.2531 |
| w9 | -0.925 | 0.8591 |
| w10 | 1.0623 | 0.311 |

|  |  |
| --- | --- |
| **B2** | Bias |
| b1 | -0.7587 |

|  |  |
| --- | --- |
| **W2** | Weight |
| w1 | -0.0069 |
| w2 | 0.1031 |
| w3 | -0.0342 |
| w4 | -0.0847 |
| w5 | 0.0509 |
| w6 | 0.1801 |
| w7 | 0.0204 |
| w8 | -0.055 |
| w9 | 0.1066 |
| w10 | -0.2359 |

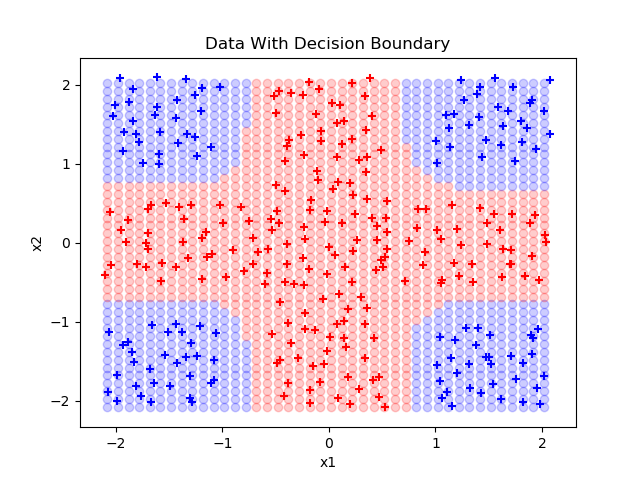
After running 1 epoch, I reset the weights to initial values and ran the algorithm. The network converged almost every time in under 100 epochs. The results are:

|  |  |
| --- | --- |
| **B1** | Bias |
| b1 | 3.2572 |
| b2 | 3.3672 |
| b3 | 3.2299 |
| b4 | 3.1794 |
| b5 | 3.315 |
| b6 | 3.4442 |
| b7 | 3.2845 |
| b8 | 3.2091 |
| b9 | 3.3707 |
| b10 | 3.0282 |

|  |  |  |
| --- | --- | --- |
| **W1** | From x1 | From x2 |
| w1 | -0.0513 | -5.3057 |
| w2 | 0.0881 | 4.8483 |
| w3 | 6.3651 | 0.1062 |
| w4 | -0.0443 | -4.7785 |
| w5 | 6.6881 | 0.1143 |
| w6 | 0.1049 | 5.9948 |
| w7 | -0.0507 | -5.2537 |
| w8 | -6.225 | -0.0655 |
| w9 | 6.413 | -0.0651 |
| w10 | 0.0527 | 3.4224 |

|  |  |
| --- | --- |
| **B2** | Bias |
| b1 | -12.6516 |

|  |  |
| --- | --- |
| **W2** | Weight |
| w1 | -3.2666 |
| w2 | -2.8687 |
| w3 | -4.7036 |
| w4 | -2.7977 |
| w5 | -4.9355 |
| w6 | -3.8123 |
| w7 | -3.2194 |
| w8 | -4.9047 |
| w9 | -5.0537 |
| w10 | -1.7356 |

Next I plotted the data points with the decision boundary along with the error per epoch: 

**+** Data Point In Class 1

**+** Data Point in Class 2

* Classified as Class 1
* Classified as Class 1

### C:\Users\hurtja\AppData\Local\Microsoft\Windows\INetCache\Content.Word\err.png

### Part B

For part B, I decided to only have 3 neurons in the hidden layer, making the network a 4:3:2 MLP. The network converged more quickly for part B than part A. The weights from the inputs to the first hidden layer after training were:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | From x1 | From x2 | From x3 | From x4 |
| w1 | 0.1032 | -0.6229 | 0.1025 | -1.2441 |
| w2 | 0.1491 | -0.4094 | 0.0462 | -1.092 |
| w3 | 0.1696 | -0.589 | 0.16 | -1.1416 |

The bias of the 3 neurons in the hidden layer were:

|  |  |
| --- | --- |
|  | Bias |
| b1 | 0.2861 |
| b2 | 1.3899 |
| b3 | 0.7905 |

The second set of weights, from the hidden layer to the output layer were:

|  |  |  |  |
| --- | --- | --- | --- |
|  | From x1 | From x2 | From x3 |
| w1 | 1.1443 | 1.2543 | 1.117 |
| w2 | -1.5434 | -1.4078 | -1.2786 |

And the bias of the neurons in the output layer:

|  |  |
| --- | --- |
|  | Bias |
| b1 | 1.8517 |
| b2 | -1.6488 |

Part B also asked us to set aside the first 100 samples from each class to test our network. After training the network to obtain the above weights and biases, I tested the network on the remaining 200 points. The testing resulted in:

Accuracy = 200/200 = 100%

## Results Analysis

The results obtained in these experiments are very similar to what I expected. I knew that both sets of data were linearly separable, so I wasn’t surprised when the network converged in under 100 epochs for both parts A and B. I was surprised however, by the difficulty of implementing the algorithm. The theoretical portion of backpropagation is not very daunting, so I expected the implementation to quickly and smoothly, especially with the many helpful libraries involved with Python. I also did not expect the network to have 100% accuracy on part B. I figured with only 800 data points for training and only 3 neurons in the hidden layer, that my accuracy would be closer to 80 or 85% at the highest. I was pleasantly surprised.

I also expected the error per epoch to be more uniform. I ran the algorithm something similar to 20 times and the error per epoch curve looked differently each time. The core curve was the same, but where it flattened was very much changed between different iterations of the algorithm. I suspect that this occurred due to the randomization of data between epochs. This also must have been affected by the value of momentum and the learning rate. When I ran the algorithm with different learning rates, the number of epochs to convergence changed with it. This change did not occur linearly but there seemed to be some direct relationship for these 2 datasets

I also really learned just how fragile the system can be. By changing the learning rate from something around .7 to something around .2, the number of iterations really jumped up, which was expected. What I also did not anticipate what the rigidness that the data needed. To extract the same number of meaningful features that can be parameterized for a large sample size is extraordinarily difficult I would think. The network cannot, for example, handle 4 features on one input and 5 on another and 10 on another. Each feature must be parameterized and be meaningful for the network to perform and converge well, which is not something I previously anticipated.

## Code

### mlp.py

The main file for the program

# Multilayer Perceptron algorithm

# implemented using backpropagation

# @author James Hurt, 2017

import pandas as pd

import argparse

import os

from pprint import pprint

from random import random

import math

# my files

from graph import \*

from calculations import \*

from constants import \*

def init():

"""

Initialize the program by getting all input data

"""

# parse command line arguments

num\_dim, num\_hidden, num\_output, partA, partB = getInputArgs()

# can't execute both parts A and B

if partA and partB:

print("Error! You cannot run both part A and B!")

exit(1)

# if the user wishes to do part A

if partA:

# set the filenames

input\_filename = "data/cross\_data.csv"

w1\_filename = "data/w1.csv"

w2\_filename = "data/w2.csv"

b1\_filename = "data/b1.csv"

b2\_filename = "data/b2.csv"

# set the sizes of the layers

num\_dim, num\_hidden, num\_output = 2, 10, 1

# read the input file

input\_file = pd.read\_csv(input\_filename, header=None)

# extract train data and labels from the input data

train\_data = input\_file.iloc[:, :-1]

labels = input\_file.iloc[:, -1:]

# if we want part B

elif partB:

# read the data file

all\_data = pd.read\_csv(

"data/Two\_Class\_FourDGaussians500.txt", sep=" ", header=None, engine='python')

# set the dims of the network

num\_dim, num\_hidden, num\_output = 4, 3, 2

# extract the training and validation data

train\_and\_validation = all\_data.iloc[:, :-1]

# extract labels

labels = all\_data.iloc[:, -1:]

# need to separate the train and validation data

# get points 100-500 and points 600-1000 as training data

train\_data = pd.concat([

train\_and\_validation.iloc[100:500, :], train\_and\_validation.iloc[600:, :]])

labels = pd.concat([labels.iloc[100:500, :], labels.iloc[600:, :]])

# get points 0-100 and 500-600 (the first 100 of each set) as validation/test data

validation\_data = pd.concat([

train\_and\_validation.iloc[:100, :], train\_and\_validation.iloc[500:600, :]])

validation\_labels = pd.concat([

labels.iloc[:100, :], labels.iloc[500:600, :]])

# convert the labels to numpy arrays of 2 dims instead of a single number

validation\_labels = label\_2D(validation\_labels)

# convert validation data to numpy

validation\_data = validation\_data.values

# set filenames for w1, w2, b1, b2

w1\_filename = "data/partB\_w1.csv"

w2\_filename = "data/partB\_w2.csv"

b1\_filename = "data/partB\_b1.csv"

b2\_filename = "data/partB\_b2.csv"

else:

# if we don't want part a or part b, then we need the filenames

# read in filenames

input\_filename = input("Enter train data filename: ")

w1\_filename = input(

"Enter weights filename from input to first hidden layer: ")

w2\_filename = input("Enter weights from hidden layer to output: ")

b1\_filename = input(

"Enter bias filename from input to first hidden layer: ")

b2\_filename = input(

"Enter bias filename from hidden layer to output: ")

# concat the current directory so we can use absolute paths

input\_filename = os.path.join(os.getcwd(), input\_filename)

w1\_filename = os.path.join(os.getcwd(), w1\_filename)

w2\_filename = os.path.join(os.getcwd(), w2\_filename)

b1\_filename = os.path.join(os.getcwd(), b1\_filename)

b2\_filename = os.path.join(os.getcwd(), b2\_filename)

# read in CSV files

input\_file = pd.read\_csv(input\_filename, header=None)

# get the train data

train\_data = input\_file.iloc[:, :-1]

# get the labels

labels = input\_file.iloc[:, -1:]

# try to actually get the data

try:

w1 = pd.read\_csv(w1\_filename, header=None)

b1 = pd.read\_csv(b1\_filename, header=None)

w2 = pd.read\_csv(w2\_filename, header=None)

b2 = pd.read\_csv(b2\_filename, header=None)

except Exception as err:

# if we error - tell the user and exit

print("Error! {}\nUnable to parse arguments and get data!".format(err))

exit(1)

# convert everythine to NumPy arrays

train\_data = train\_data.values

labels = labels.values if not partB else label\_2D(labels)

w1 = w1.values

w2 = w2.values

# flatten into simgle dim because b1 and b2 should always be single dim

b1 = b1.values.flatten()

b2 = b2.values.flatten()

# run the algorithm

if partA: # part A only runs a single epoch

# run the alg

w1\_new, w2\_new, b1\_new, b2\_new, avg\_error\_energy = epoch(

train\_data, labels, w1, w2, b1, b2)

# print the values

print\_results(w1\_new, w2\_new, b1\_new, b2\_new, avg\_error\_energy)

# wait for user input

input("\n\nPress [Enter] to train network...\n")

# run the entire algorithm for the next part of part A

w1, w2, b1, b2, error\_per\_epoch = run(

train\_data, labels, w1, w2, b1, b2)

# allow user to see data

graph\_init(error\_per\_epoch, train\_data, labels, w1, w2, b1, b2)

elif partB:

# train the network

w1, w2, b1, b2, error = run(train\_data, labels, w1, w2, b1, b2)

# wait for user to ok testing

input("\n\nPress [Enter] to test network...\n")

print("Validating Data...")

# test the data

accuracy = validate(validation\_data, validation\_labels, w1, w2, b1, b2)

# display the accuracy

print("Accuracy: {}".format(accuracy))

else:

# run the entire algorithm

run(train\_data, labels, w1, w2, b1, b2)

def validate(validation\_data, validation\_labels, w1, w2, b1, b2):

"""

Validate with validation data

"""

# number correct

num\_correct = 0

# iterate through validation data and show it to the network

for i, datapoint in enumerate(validation\_data):

# show to hidden layer

hidden = show\_to\_layer(datapoint, w1, b1)

# show to output layer

output = show\_to\_layer(hidden, w2, b2)

# values aren't exactly 1 and 0, need to round

output = np.around(output)

# if we get it right, then inc num\_correct

if np.array\_equal(validation\_labels[i], output):

num\_correct += 1

# accuracy is the num correct over the total

a = float(num\_correct) / float(len(validation\_data))

return a

def label\_2D(labels):

"""

Take in labels of 1 and 2 and turn them into a 2D vector of (1,0) and (0,1)

"""

# turn labels into a numpy array of 1 dimension

labels = labels.values.flatten()

# create an empty array

new\_labels = np.empty([len(labels), 2])

# iterate through labels

for i, l in enumerate(labels):

# make this value [0,1] if the current label is 0, otherwise make this label [1,0]

new\_labels[i] = np.array(

[0, 1]) if l == 0 else np.array([1, 0])

# return 2D labels

return new\_labels

def print\_results(w1, w2, b1, b2, avg\_error\_energy):

"""

Prints the results given NumPy arrays

"""

print("----------------------\n\t\tW1\n----------------------")

pprint(np.around(w1, decimals=4).tolist())

print("----------------------\n\t\tW2\n----------------------")

pprint(np.around(w2, decimals=4).tolist(), width=1)

print("----------------------\n\t\tB1\n----------------------")

pprint(np.around(b1, decimals=4).tolist(), width=1)

print("----------------------\n\t\tB2\n----------------------")

pprint(np.around(b2, decimals=4).tolist(), width=1)

print(

"----------------------\n\t\tERROR\n----------------------\nAverage Error Energy: {:10.4f}".format(

avg\_error\_energy))

def getInputArgs():

"""

Get all arguments from the command line

"""

# create the argument parser

parser = argparse.ArgumentParser()

# the number of features / dimensions

parser.add\_argument('num\_dim', help='The number of features')

# the number of nuerons in hidden layer

parser.add\_argument(

'num\_hidden', help='Number of neurons in the hidden layer')

# the number of output nodes

parser.add\_argument(

'num\_output', help='Number of neurons in the output layer')

# default to part a

parser.add\_argument(

'-a', '--partA', action="store\_true", help="Run part A")

# part B

parser.add\_argument(

'-b', '--partB', action="store\_true", help="Run part B")

# parse the arguments

args = vars(parser.parse\_args())

# store the directories as variables

num\_dim, num\_hidden, num\_output = args["num\_dim"], args["num\_hidden"], args["num\_output"]

# check for part A

partA = True if args["partA"] else False

# check for part B

partB = True if args["partB"] else False

# return

return num\_dim, num\_hidden, num\_output, partA, partB

def run(training\_data, desired\_output, w1, w2, b1, b2):

"""

Run the algorithm with the given parameters

"""

# the errors of each epoch

error = []

# iteration number

i = 0

# previous error to calc next error

prev\_error = 1

# run forever!

while True:

# randomize data

training\_data, desired\_output = randomize\_data(

training\_data, desired\_output)

# run an epoch

w1, w2, b1, b2, avg\_error = epoch(

training\_data, desired\_output, w1, w2, b1, b2)

# check termination condition

if avg\_error < TERMINATION\_THRESHOLD:

break

# incrememnt iterate counter

i += 1

# add this epochs error to the list

error.append(avg\_error)

# calc the change in diff

diff = prev\_error - avg\_error

# calc the percent error

diff = diff / prev\_error \* 100

# absolute value

diff = diff if diff >= 0 else -diff

# print the iteration num, the percent change, and the average error

print("Epoch Number: {:6d} \t{:>15} {:.7f}\tPercent Change: {:.4f}".format(

i, "Average Error: ", avg\_error, diff

))

# move the avg to prev

prev\_error = avg\_error

# once we've terminated, print the results

print\_results(w1, w2, b1, b2, avg\_error)

# return

return w1, w2, b1, b2, error

def randomize\_data(a, b):

"""

Randomize a, b so that the order shown to the

network is random

"""

# create empty arrays

shuffled\_a = np.empty(a.shape, dtype=a.dtype)

shuffled\_b = np.empty(b.shape, dtype=b.dtype)

# get a permutation of array a

permutation = np.random.permutation(len(a))

# go throgh the permutation and switch values in both labels and data

for old\_index, new\_index in enumerate(permutation):

shuffled\_a[new\_index] = a[old\_index]

shuffled\_b[new\_index] = b[old\_index]

# return the shuffled arrays

return shuffled\_a, shuffled\_b

def epoch(training\_data, desired\_output, w1, w2, b1, b2):

"""

Run a single epoch throught network with given params

"""

# initiate the avg error

avg\_error = 0.

# assign prev to current for first iteration

previous\_w1, previous\_w2, previous\_b1, previous\_b2 = w1, w2, b1, b2

# run an epoch

for i, datapoint in enumerate(training\_data):

# show to first hidden layer

first\_layer\_output = show\_to\_layer(datapoint, w1, b1)

# show to output layer

output = show\_to\_layer(first\_layer\_output, w2, b2)

# error

er = calc\_error(output, desired\_output[i])

# add this error to the total error

avg\_error += er

# backpropoate

next\_w1, next\_w2, next\_b1, next\_b2 = backpropagate(datapoint,

output, first\_layer\_output, desired\_output[i], w1, w2, b1,

b2, previous\_w1, previous\_w2)

# set prev to current

previous\_w1, previous\_w2, previous\_b1, previous\_b2 = w1, w2, b1, b2

# set current to next

w1, w2, b1, b2 = next\_w1, next\_w2, next\_b1, next\_b2

# avg error = 1/2K \* total error

avg\_error = avg\_error / (2. \* len(training\_data))

# return

return w1, w2, b1, b2, avg\_error

def backpropagate(datapoint, output, output\_layer1, label, w1, w2, b1, b2, previous\_w1, previous\_w2):

"""

Backpropagate the error

w(k+1) = w(k) + B(w(k) - w(k-1)) + A(delta)(output)

"""

# output layer

output\_deltas = np.empty(len(w2))

w2\_new = np.empty([len(w2), len(w2[0])])

for i, (neuron, prev\_neuron) in enumerate(zip(w2, previous\_w2)):

# holder variable

new\_nueron\_w = np.empty(len(neuron))

# calc the delta: delta = e \* fiprime(v)

v = calc\_v(output\_layer1, neuron, b2[i])

prime = fi\_prime(v)

e = label[i] - output[i]

delta = prime \* e

output\_deltas[i] = delta

# get the learning term

learn\_term = ALPHA \* delta \* output\_layer1[i]

# adjust the bias

b2[i] = adjust\_b(b2[i], delta)

for j, (weight, prev\_weight) in enumerate(zip(neuron, prev\_neuron)):

# calc momentum term

momentum\_term = BETA \* (weight - prev\_weight)

# calculate the difference

diff = momentum\_term + learn\_term

# calculate the new weight

new\_w = weight + diff

# store this result

new\_nueron\_w[j] = new\_w

# store the result

w2\_new[i] = new\_nueron\_w

# set this to w2

w2 = w2\_new

# hidden layer

w1\_new = np.empty([len(w1), len(w1[0])])

for i, (neuron, prev\_neuron) in enumerate(zip(w1, previous\_w1)):

# holder variable

new\_nueron\_w = np.empty(len(neuron))

# calc the delta = fiprime \* summation(delta\_output \* w)

v = calc\_v(datapoint, neuron, b1[i])

prime = fi\_prime(v)

summation = 0

# iterate through the deltas of the output layer

for j, d in enumerate(output\_deltas):

# multiply the delta \* the weight connecting to that neuron

summation += d \* w2[j][i]

delta = prime \* summation

# adjust the bias

b1[i] = adjust\_b(b1[i], delta)

for j, (weight, prev\_weight) in enumerate(zip(neuron, prev\_neuron)):

# calc momentum term

momentum\_term = BETA \* (weight - prev\_weight)

# calc learn term

learn\_term = ALPHA \* delta \* datapoint[j]

# calculate the difference

diff = momentum\_term + learn\_term

# calculate the new weight

new\_w = weight + diff

# store this result

new\_nueron\_w[j] = new\_w

# store the result

w1\_new[i] = new\_nueron\_w

# set this to w2

w1 = w1\_new

# return new parameters

return w1, w2, b1, b2

def show\_to\_layer(inputs, weights, biases):

"""

Take in the input, weights, and bias and show an input

to the layer specified by the weights and biases

"""

# rename inputs

w1 = weights

training\_data = inputs

b1 = biases

num\_neurons = len(w1)

# create the array to hold the output of this layer

next\_layer\_input = np.empty(num\_neurons)

# go through each neuron in this layer

for j, weights in enumerate(w1):

# v = wTx + b

v = np.dot(inputs, weights) + b1[j]

# output is the activation function

output = 1 / (1 + math.e\*\*(-v))

# put in the array

next\_layer\_input[j] = output

# return the output of this layer

return next\_layer\_input

if \_\_name\_\_ == "\_\_main\_\_":

init()

### graph.py

File with helper functions to graph the output for part A

from matplotlib import pyplot as plt

import numpy as np

import os

import math

from mlp import show\_to\_layer

def graph\_init(error\_per\_epoch, train\_data, labels, w1, w2, b1, b2):

"""

Allow user to see graphs

"""

# the input

num = 0

# sentinal

while num != 3:

# get user input

num = int(

input("Would you like to view: \n1. Error Per Epoch\n2. Data\n3. Exit\n>"))

if num == 1:

# grph error per epoch

graph\_error\_per\_epoch(error\_per\_epoch)

elif num == 2:

# graph the data

graph\_data\_with\_solution(train\_data, labels, w1, w2, b1, b2)

def graph\_error\_per\_epoch(error\_per\_epoch):

"""

Graph the error per epoch

"""

# plot the graph

plt.plot(range(len(error\_per\_epoch)), error\_per\_epoch)

# set title

plt.title("Error Per Epoch")

# set labels

plt.ylabel("Error")

plt.xlabel("Number of Epochs")

# show the graph

plt.show()

def graph\_data\_with\_solution(train\_data, labels, w1, w2, b1, b2):

"""

Graph the data

"""

# find the largest x value and largest y value

biggest\_x = -1

biggest\_y = -1

for x, y in train\_data:

if x > biggest\_x:

biggest\_x = x

if y > biggest\_y:

biggest\_y = y

# create arrays going from - biggest x/y to + biggest x/y with interval .1

x\_array = np.arange(-biggest\_x, biggest\_x, .1)

y\_array = np.arange(-biggest\_y, biggest\_y, .1)

# iterate through the arrays

for i in x\_array:

for j in x\_array:

# create a datapoint

datapoint = np.array([i, j])

# show to hidden layer

hidden = show\_to\_layer(datapoint, w1, b1)

# show to output layer

output = show\_to\_layer(hidden, w2, b2)[0]

# values aren't exactly 1 and 0, need to round

output = np.around(output)

# plot the point

plt.scatter(i, j, c="red" if output ==

0 else "blue", alpha=.2)

for (x, y), label in zip(train\_data, labels):

# plot the data, altering color based on label

plt.scatter(x, y, c=("red" if label == 0 else "blue"),

alpha=1, marker="+")

# set properties

plt.title("Data With Decision Boundary")

plt.xlabel("x1")

plt.ylabel("x2")

plt.legend()

# show the graph

plt.show()

### calculations.py

The file that holds helper functions for calculations, such as the activation function and the induced local field

import numpy as np

import math

from constants import \*

def calc\_v(inputs, weights, bias):

"""

Calculate v, the input vector

v = wTp + b

"""

return np.dot(inputs, weights) + bias

def fi\_prime(v):

"""

Return the value of the activation function's derivative

"""

# calc fi

f = fi(v)

# fi \* 1 - fi

return f \* (1 - f)

def adjust\_b(bias, delta):

"""

Adjust the bias

bias += ALPHA \* 1 \* delta

"""

return bias + (ALPHA \* delta)

def fi(v):

"""

Define the activation function and return fi of v

"""

# define the sigmoid and return the value

denom = 1 + math.e \*\* (-1 \* v)

val = 1 / denom

return val

def calc\_error(output, desired\_output):

"""

Calculate the error at the output layer

"""

er = 0

# iterate throuh all output neurons

for y, d in zip(output, desired\_output):

# add this error to total error

er += (y - d)\*\*2

return er