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 $v = w^T x$ 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
    a. Choose initial weights and bias
    b. Load training data
    c. Choose activation function
    d. Choose constants: Alpha, Beta, and Termination Threshold
2. Present point p_i to the network and obtain the output
3. Calculate error of p_i
4. Backpropagate
    a. Calculate the momentum term using predefined momentum constant and the previous change in w_{ji}
    b. Calculate the delta for neuron j
        i. For an output neuron, multiply the derivative of the activation function at the induced local field by error of the neuron
        ii. 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
    c. Multiply the delta for neuron j with the predefined constant for learning rate as well the value of the source neuron for weight w_{ji}

d. Add the value calculated in part c to the current value of w_{ji} and assign that value as the next w_{ji}

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:

W1	From x_1	From x_2
w_1	0.4292	-1.1591
w_2	0.4619	0.3795
w_3	0.6854	-0.2006
w_4	0.0679	-0.2199
w_5	0.1865	0.1026
w_6	-0.0906	0.9862
w_7	0.794	-0.6615
w_8	-0.6719	0.2531
w_9	-0.925	0.8591
w_{10}	1.0623	0.311

B1	Bias
b_1	-0.1495
b_2	0.9248
b_3	0.4322
b_4	-0.1462
b_5	-0.7105
b_6	-0.5103
b_7	-0.705
b_8	-0.9313
b_9	-0.4693
b_{10}	1.1784

W2	Weight
w_1	-0.0069
w_2	0.1031
w_3	-0.0342
w_4	-0.0847
w_5	0.0509
w_6	0.1801
w_7	0.0204
w_8	-0.055
w_9	0.1066

B2	Bias
b_1	-0.7587

w_{10}	-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:

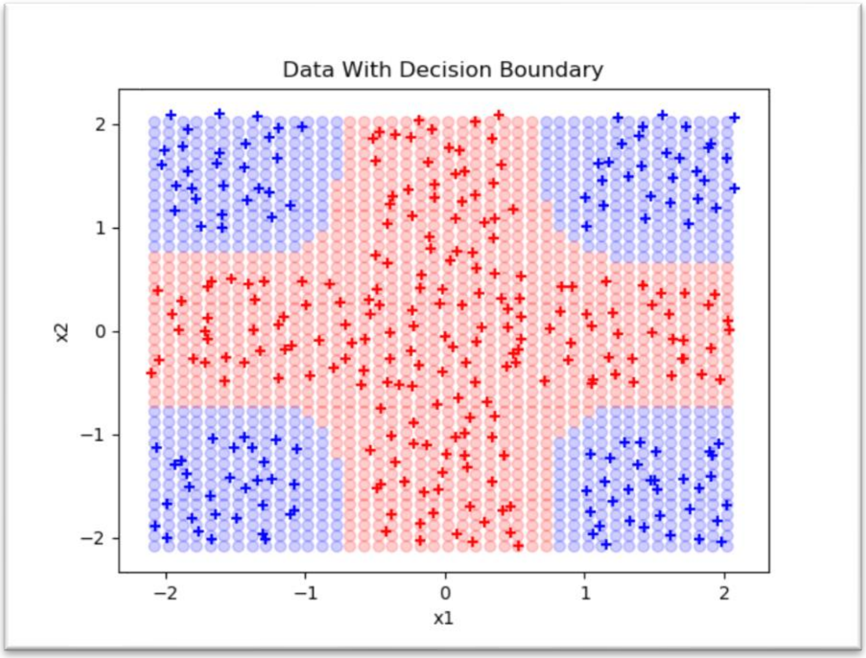
W1	From x_1	From x_2
w_1	-0.0513	-5.3057
w_2	0.0881	4.8483
w_3	6.3651	0.1062
w_4	-0.0443	-4.7785
w_5	6.6881	0.1143
w_6	0.1049	5.9948
w_7	-0.0507	-5.2537
w_8	-6.225	-0.0655
w_9	6.413	-0.0651
w_{10}	0.0527	3.4224

B1	Bias
b_1	3.2572
b_2	3.3672
b_3	3.2299
b_4	3.1794
b_5	3.315
b_6	3.4442
b_7	3.2845
b_8	3.2091
b_9	3.3707
b_{10}	3.0282

W2	Weight
w_1	-3.2666
w_2	-2.8687
w_3	-4.7036
w_4	-2.7977
w_5	-4.9355
w_6	-3.8123
w_7	-3.2194
w_8	-4.9047
w_9	-5.0537
w_{10}	-1.7356

B2	Bias
b_1	-12.6516

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

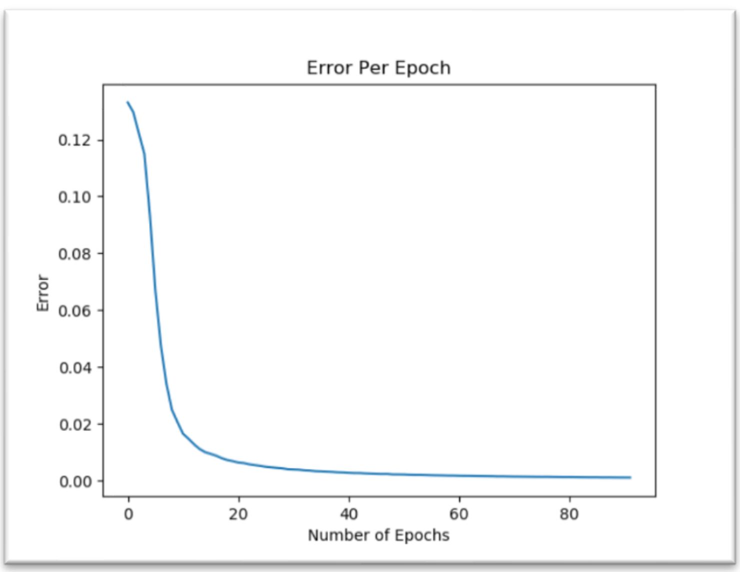


+ Data Point In Class 1

+ Data Point in Class 2

● Classified as Class 1

● Classified as Class 1



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 x_1	From x_2	From x_3	From x_4
w_1	0.1032	-0.6229	0.1025	-1.2441
w_2	0.1491	-0.4094	0.0462	-1.092

w_3	0.1696	-0.589	0.16	-1.1416

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

	Bias
b_1	0.2861
b_2	1.3899
b_3	0.7905

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

	From x_1	From x_2	From x_3
w_1	1.1443	1.2543	1.117
w_2	-1.5434	-1.4078	-1.2786

And the bias of the neurons in the output layer:

	Bias
b_1	1.8517
b_2	-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)
```

```python
        # 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
        # incremennt 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)
```

```python
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
```