Project Report

Artificial Intelligence

Raymond Zhu

923008555

Statement of the Problem

The problem to this project is to develop a neural network that can distinguish between capital letters and attempt to identify these letters with some noise (bit flipping). The resources for this project is a set of hex codes used for a 5x7 bit matrix and a back-propagation algorithm from figure 18.24 in *Artificial Intelligence: A Modern Approach 3rd*. The project must implement the back-propagation algorithm using any programming language.

Restrictions and Limitations

This project requires a minimum of 2 layers in the neural network, a set of inputs that fan out to the input layer and an output layer. In addition to this restriction, the neural network needs to be accurate; an accuracy of approximately 90% is acceptable. The project should also include user input that accepts a 35-character string to test the accuracy of the neural network.

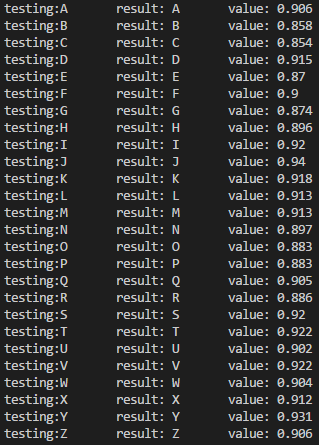
Explanation of Approach

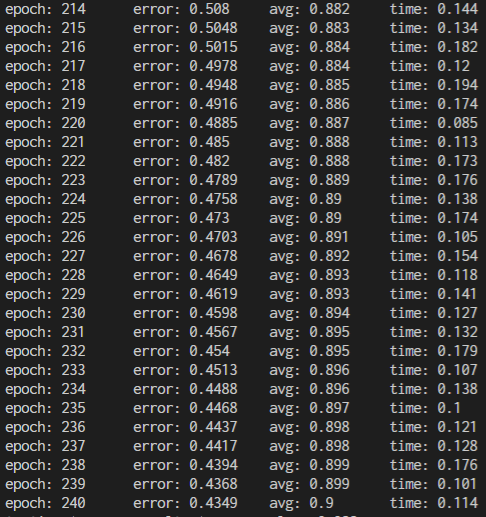
This project uses python3.6 as the programming language of choice. Although C++ may be a better language for training and testing, the convenience of developing in python was chosen over the latter. 3 layers are used in this project, consisting of an input, hidden, and output layer with 35, 35, and 26 neurons respectively. Each letter for training is converted into a list of binary values from the given hex font header file. Since tanh and sigmoid standardize the weights of the neural network within two different ranges, sigmoid is only used for the output layer whereas tanh is used for all the other layers. This makes sense because the output layer consists of values between [0,1], and sigmoid ranges between [0,1].

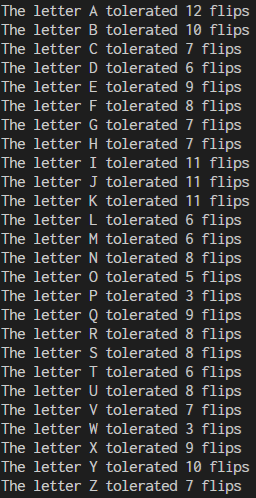
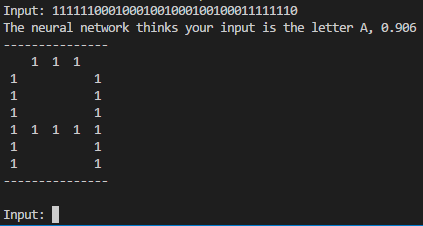
The learning rate of each layer is dependent on the number of neurons. This project uses the formula .

Using the backpropagation algorithm, this project uses a gradient descent approach where weights are updated after each example, with a total of 26 examples (the alphabet). Every cycle of 26 letters is considered an epoch, and the project runs x # of iterations of the backpropagation algorithm for the training phase. The number of epochs varies because the training stops when the average output of all the letters is greater than or equal to 90%.

Sample Run

In the following screenshots, the project runs two tests after its training. The first screenshot shows the iteration number and the calculated total error of the neural network. The following image is a simple test to check whether the neural network recognizes each letter. The value of the output value is shown to verify that the accuracy of the neural network is above 90%. Afterwards, the project runs a bit flip test against the trained neural network and the results are tabulated below. At the end of training and testing the program will prompt for user input to test the neural network manually and graphically displays the inputted letter to the user

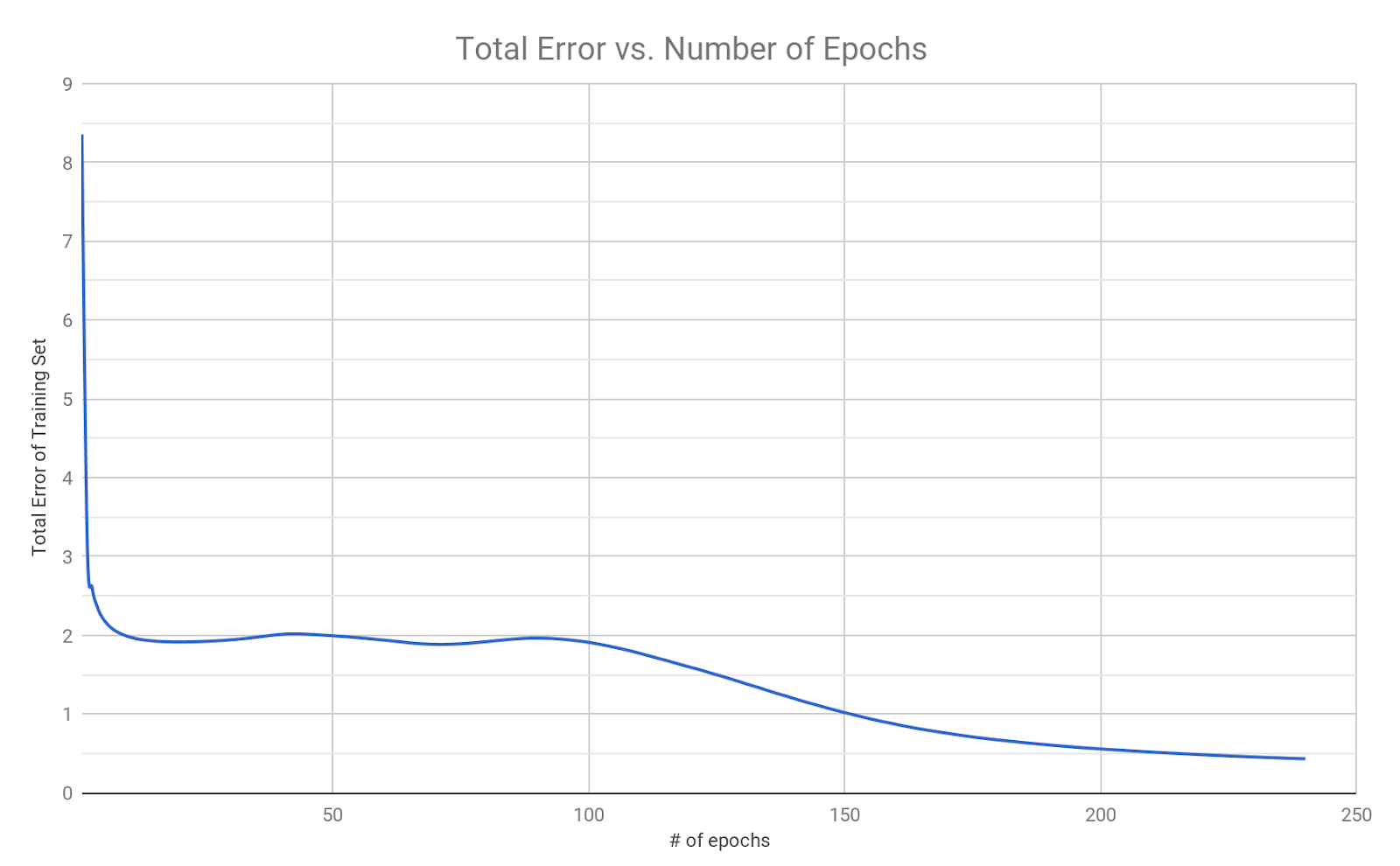




Results and Analysis

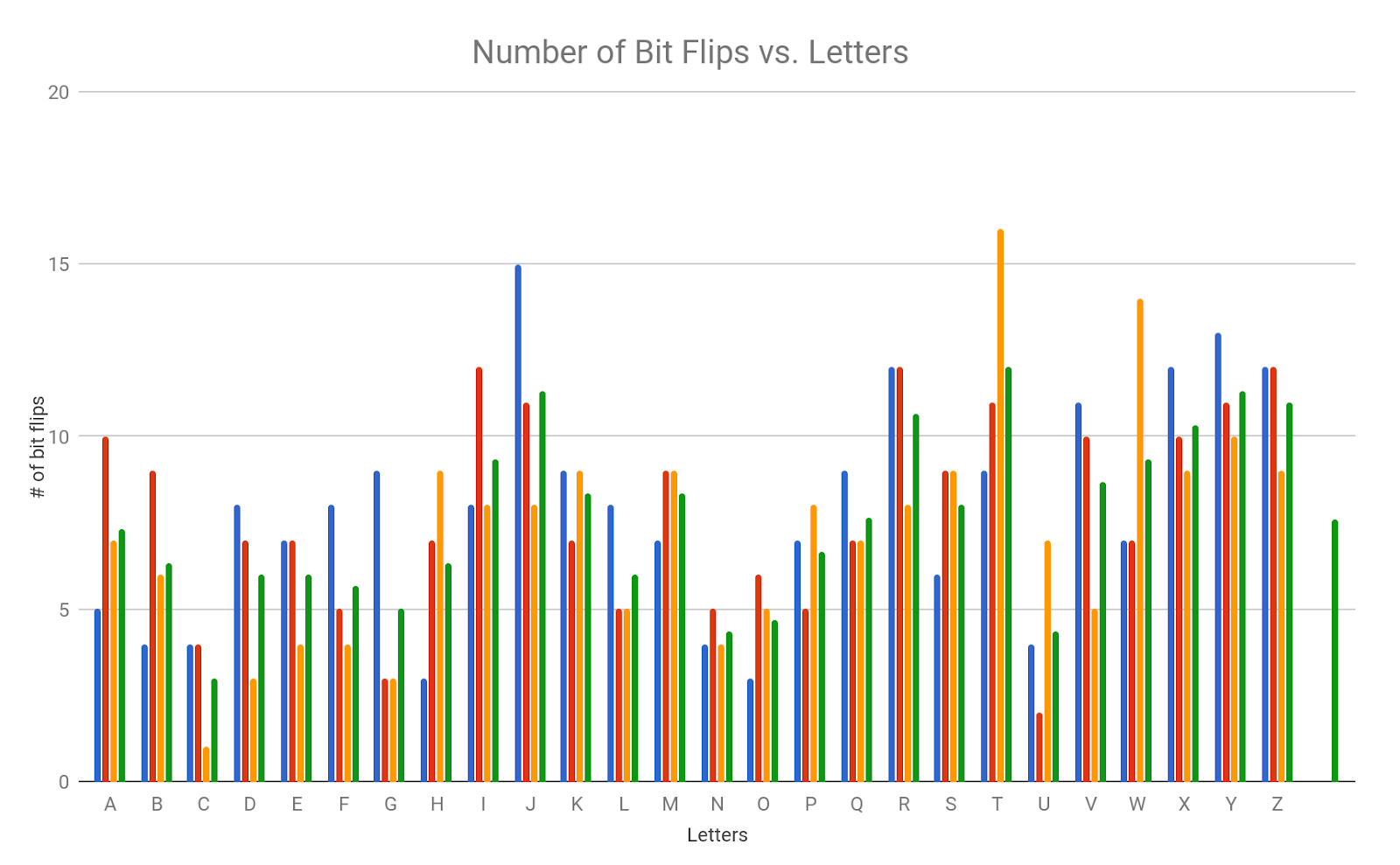
The total error of every epoch was used to graph the learning rate of the neural network. Since each epoch consists of all 26 letters, the calculated total error is the absolute value of the sum of all the error vectors of each letter divided by 26. The values were saved into a text file and graphed in a spreadsheet in the figure below

As it can be seen in the following graph, the initial total error is high because of random weights. However, as the neural network undergoes 1000 epochs, the total error eventually drops below 1. Between epoch 0 to 100, it can be noticed on the graph that there is a slight bump. This may well be the case of the neural network reaching a local minimum, where additional backpropagation iterations later adjusted the network to reach a global minimum.



For the bit flip test, the trained neural network ran against the bit-flip test three times. Since the flips are random, it can be seen from the results that the tolerance varies for each test. Blue, red, and orange represent each test and the green bar represents the average tolerance of the three tests. From the averages in the graph below, each letter can tolerate roughly around 5-bit flips. The very last column indicates the average of all the letters, tolerating an average of 8 flips. A table has also been included to tabulate the tolerances.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Letter | Test 1 | Test 2 | Test 3 | Average |
| A | 5 | 10 | 7 | 7.3 |
| B | 4 | 9 | 6 | 6.3 |
| C | 4 | 4 | 1 | 3.0 |
| D | 8 | 7 | 3 | 6.0 |
| E | 7 | 7 | 4 | 6.0 |
| F | 8 | 5 | 4 | 5.7 |
| G | 9 | 3 | 3 | 5.0 |
| H | 3 | 7 | 9 | 6.3 |
| I | 8 | 12 | 8 | 9.3 |
| J | 15 | 11 | 8 | 11.3 |
| K | 9 | 7 | 9 | 8.3 |
| L | 8 | 5 | 5 | 6.0 |
| M | 7 | 9 | 9 | 8.3 |
| N | 4 | 5 | 4 | 4.3 |
| O | 3 | 6 | 5 | 4.7 |
| P | 7 | 5 | 8 | 6.7 |
| Q | 9 | 7 | 7 | 7.7 |
| R | 12 | 12 | 8 | 10.7 |
| S | 6 | 9 | 9 | 8.0 |
| T | 9 | 11 | 16 | 12.0 |
| U | 4 | 2 | 7 | 4.3 |
| V | 11 | 10 | 5 | 8.7 |
| W | 7 | 7 | 14 | 9.3 |
| X | 12 | 10 | 9 | 10.3 |
| Y | 13 | 11 | 10 | 11.3 |
| Z | 12 | 12 | 9 | 11.0 |



Conclusions

In conclusion, the results of this project show that a neural network can be used to solve pattern recognition problems. Using random weights, a neural network can adjust its weights through backpropagation and learn the correct weights that will satisfy the output layer. It is shown that because of random weights, every set of trained data will always be different when compared to another set. Although the accuracy of different sets is similar at approximately >90%, the bit flip test shows that results can vary when the environment is stochastic, and nondeterministic. This project taught me how to develop a simple neural network and many lessons were learned throughout the development process. These many lessons include use of efficient data structures to hold large amounts of data and understanding the problem fully before attempting to implement it.

Future Research

One simple change could be to split the training and testing into two separate script files to reduce the time spent training. Right now, the only way to test data is to retrain the neural network and test the results. Alternatively, one can train data and store it as an object file. Another script can then load such objects and test the results. As shown above, since results vary, and there is a lot of randomness to testing, using the same training data can potentially yield many different outcomes.

Instruction to Run Project

To run this project, please make sure that python 3.6 is installed and fully functionally. An external library numpy is needed to properly run this code.

$ python3.6 project.py // this will run the code

$ Input: // this will be displayed after training and tests are complete. A string of the letter font in binary of 35 characters is accepted as input

The result of this code will create 3 files, training\_data.txt, testing\_data.txt, and bitflip.txt.

Code

|  |  |  |
| --- | --- | --- |
| |  | | --- | | # Raymond Zhu # 923008555 # CSCE 420 # Due: April 23, 2018 # project.py  import math import random import time import operator import copy import sys import numpy as np  #   ASCII FONT A = ["0x7E", "0x11", "0x11", "0x11", "0x7E"] B = ["0x7F", "0x49", "0x49", "0x49", "0x36"] C = ["0x3E", "0x41", "0x41", "0x41", "0x22"] D = ["0x7F", "0x41", "0x41", "0x22", "0x1C"] E = ["0x7F", "0x49", "0x49", "0x49", "0x41"] F = ["0x7F", "0x09", "0x09", "0x01", "0x01"] G = ["0x3E", "0x41", "0x41", "0x51", "0x32"] H = ["0x7F", "0x08", "0x08", "0x08", "0x7F"] I = ["0x00", "0x41", "0x7F", "0x41", "0x00"] J = ["0x20", "0x40", "0x41", "0x3F", "0x01"] K = ["0x7F", "0x08", "0x14", "0x22", "0x41"] L = ["0x7F", "0x40", "0x40", "0x40", "0x40"] M = ["0x7F", "0x02", "0x04", "0x02", "0x7F"] N = ["0x7F", "0x04", "0x08", "0x10", "0x7F"] O = ["0x3E", "0x41", "0x41", "0x41", "0x3E"] P = ["0x7F", "0x09", "0x09", "0x09", "0x06"] Q = ["0x3E", "0x41", "0x51", "0x21", "0x5E"] R = ["0x7F", "0x09", "0x19", "0x29", "0x46"] S = ["0x46", "0x49", "0x49", "0x49", "0x31"] T = ["0x01", "0x01", "0x7F", "0x01", "0x01"] U = ["0x3F", "0x40", "0x40", "0x40", "0x3F"] V = ["0x1F", "0x20", "0x40", "0x20", "0x1F"] W = ["0x7F", "0x20", "0x18", "0x20", "0x7F"] X = ["0x63", "0x14", "0x08", "0x14", "0x63"] Y = ["0x03", "0x04", "0x78", "0x04", "0x03"] Z = ["0x61", "0x51", "0x49", "0x45", "0x43"]  # List of all the letters containing their hex codes Alphabet = [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z] # List of characters to help assign index values to letters. A-Z <=> 0-25 Characters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'] # The batch that will be used when training the neural network listOfExamples = []  class Example:    def \_\_init\_\_(self, x, y):        self.input\_vector = x        self.output\_vector = y           input\_vector = []    output\_vector  = []  class Network:    def \_\_init\_\_(self, l, i, h, o):        self.layers = l        self.inputSize = i        self.hiddenSize = h        self.outputSize = o             inputSize = 0    hiddenSize = 0    outputSize = 0    layers = 0    wout = []    whidden = []  def HEX\_TO\_BINARY(list):    # https://stackoverflow.com/questions/1425493/convert-hex-to-binary    binary\_list = []    scale = 16 ## equals to hexadecimal    num\_of\_bits = 7    for x in list:        binary\_list.append(bin(int(x, scale))[2:].zfill(num\_of\_bits))     return binary\_list  def SIGMOID\_ACTIVATION\_FUNCTION(x):    return 1 / (1 + math.exp(-x))  def TANH\_ACTIVATION\_FUNCTION(x):    return math.tanh(x)  def LOGISTIC\_SIGMOID(x):    # the derivative for sigmoid activation function    return SIGMOID\_ACTIVATION\_FUNCTION(x) \* (1-SIGMOID\_ACTIVATION\_FUNCTION(x))  def HYPERBOLIC\_TANGENT\_SIGMOID(x):    # the derivative for tanh activation function    return 1 - TANH\_ACTIVATION\_FUNCTION(x)\*\*2  def BACK\_PROP\_LEARNING(examples, network):    # initalize error arrays    errors\_output = [0 for x in range(network.outputSize)]    errors\_hidden = [0 for x in range(network.hiddenSize)]    epoch = 0       while True:        # initializing heuristics        sTime = time.time()        totalError = 0.0        letterAccuracy = []        for x in range(0, len(examples)):            weighted\_sum = []            output\_actual = []            input\_activations = []            hidden\_activations = []            hidden\_weighted\_sum = []            # forward propagation            for i in examples[x].input\_vector:                input\_activations.append(i)            for j in range(0, network.hiddenSize):                # weights array is arranged in the from [w11, w12, w13, ... , wij]                # the list is divided into j sections each containing i weights                ws = 0                for i in range(0, network.inputSize):                    # for each neuron in the hidden layer, get the sum of all its inputs                    # i\*network.hiddenSize+j = w1j, w2j, w3j ... for each j neuron in the hidden layer                    ws += (network.whidden[i\*network.hiddenSize+j] \* input\_activations[i])                hidden\_weighted\_sum.append(round(ws,3))                hidden\_activations.append(round(TANH\_ACTIVATION\_FUNCTION(ws),3))            for j in range(0, network.outputSize):                # weights array is arranged in the from [w11, w12, w13, ... , wij]                # the list is divided into j sections each containing i weights                ws = 0                for i in range(0, network.hiddenSize):                    # for each neuron in the output layer, get the sum of all its connections                    # i\*network.outputSize+j = w1j, w2j, w3j ... for each j neuron in the output layer                    ws += (network.wout[i\*network.outputSize+j] \* hidden\_activations[i])                weighted\_sum.append(round(ws,3))                output\_actual.append(round(SIGMOID\_ACTIVATION\_FUNCTION(ws),3))            # backward propagation               for j in range(0, network.outputSize):                # error calculation of output layer                errors\_output[j] = LOGISTIC\_SIGMOID(weighted\_sum[j]) \* (examples[x].output\_vector[j]-output\_actual[j])            for i in range(0, network.hiddenSize):                error = 0                for j in range(0, network.outputSize):                    # for each neuron in the hidden layer, get the sum of all its connections                    # i\*network.outputSize+j = wi1, wi2, wi3, ... for each i neuron in the hidden layer                    error += network.wout[i\*network.outputSize+j] \* errors\_output[j]                errors\_hidden[i] = HYPERBOLIC\_TANGENT\_SIGMOID(hidden\_weighted\_sum[i])\*error            # update weights               counter = 0            for w in range(0, len(network.wout)):                # need a counter to get the position of each i neuron's jth index for j layer's error array value                # math.floor() is used to get the correct index for each ith node relative to the size of weights array                network.wout[w] = network.wout[w] + 1/(math.sqrt(network.hiddenSize)) \* hidden\_activations[math.floor(w/network.outputSize)] \* errors\_output[counter]                counter += 1                if counter == network.outputSize:                    counter = 0            counter = 0            for w in range(0, len(network.whidden)):                # need a counter to get the position of each i neuron's jth index for j layer's error array value                # math.floor() is used to get the correct index for each ith node relative to the size of weights array                network.whidden[w] = network.whidden[w] + 1/(math.sqrt(network.inputSize)) \* input\_activations[math.floor(w/network.hiddenSize)] \* errors\_hidden[counter]                counter += 1                 if counter == network.hiddenSize:                    counter = 0            # Subtracting a list with a list and taking the sum of the absolute value of that list               totalError = totalError + sum(list(map(abs, list(map(operator.sub, examples[x].output\_vector, output\_actual)))))            letterAccuracy.append(output\_actual[x])         # totalError is a sum of all 26 letters' errors, need to calculate the average totalError        totalError = round(totalError/len(examples), 4)        averageAccuracy = round(sum(letterAccuracy)/26.0, 3)        print("epoch: " + str(epoch) + "\terror: " + str(totalError) + "\t avg: " + str(averageAccuracy) + "\ttime: " + str(round(time.time()-sTime,3)))        log = open("training\_data.txt", "a")        log.write(str(totalError)+"\n")         epoch += 1        if averageAccuracy >= 0.9:            break    return network  def test(example, network):    # same code used in back propagation    weighted\_sum = []    output\_actual = []    input\_activations = []    hidden\_activations = []    hidden\_weighted\_sum = []    for i in example.input\_vector:        input\_activations.append(i)    # forward propagation    for j in range(0, network.hiddenSize):        ws = 0        for k in range(0, network.inputSize):            ws += (network.whidden[k\*network.hiddenSize+j] \* input\_activations[k])        hidden\_weighted\_sum.append(round(ws,3))        hidden\_activations.append(round(TANH\_ACTIVATION\_FUNCTION(ws),3))    for k in range(0, network.outputSize):        ws = 0        for l in range(0, network.hiddenSize):            ws += (network.wout[l\*network.outputSize+k] \* hidden\_activations[l])        weighted\_sum.append(round(ws,3))        output\_actual.append(round(SIGMOID\_ACTIVATION\_FUNCTION(ws),3))       return output\_actual  def flip(example, flippedSet):    # take in a letter and flip 0 to 1 or 1 to 0 in it's input\_vector    # add the index of the flipped bit into a set flippedSet to ensure every flip is a different index    randomNum = random.randint(0, 34)    if randomNum in flippedSet:        while randomNum in flippedSet:            randomNum = random.randint(0, 34)    flippedSet.add(randomNum)     if example.input\_vector[randomNum] == 0:        example.input\_vector[randomNum] = 1    else:        example.input\_vector[randomNum] = 0       return example, flippedSet  def display(example):    # prints out what the input\_vector looks like    \_matrix = []    for k in range(0, 5):        row = []        for j in range(0, 7):            row.append(example.input\_vector[k\*7+j])        \_matrix.append(row)    matrix = np.rot90(np.matrix(\_matrix))      print("---------------")     for r in range(0 , 7):        row = ""        for c in range(0, 5):            if matrix[r, c] == 0:                row = row + "   "            else:                row = row + " 1 "        print(row)     print("---------------")     return None  for x in range(0, len(Alphabet)):    training\_output = []    training\_input = []    training\_input\_bin = HEX\_TO\_BINARY(Alphabet[x])     for i in training\_input\_bin:        # append each bit into a list for a letter's input\_vector        for j in list(i):            training\_input.append(int(j))    for y in range(0, 26):        # creates the output\_vector        if y == x:            # since the for loop runs through A-Z, each x represents A-Z            training\_output.append(1)            continue        training\_output.append(0)           listOfExamples.append(Example(training\_input, training\_output))  random.seed(time) ASCII\_NN = Network(2, 35, 35, 26) for w in range(0, ASCII\_NN.inputSize\*ASCII\_NN.hiddenSize):    # generate random weights for input/hidden layer    rw = random.uniform(-0.1, 0.1)    while rw == 0.0:        rw = random.uniform(-0.1, 0.1)    ASCII\_NN.whidden.append(round(rw,3)) for w in range(0, ASCII\_NN.hiddenSize\*ASCII\_NN.outputSize):    # generate random weights for hidden/output layer    rw = random.uniform(-0.1, 0.1)    while rw == 0.0:        rw = random.uniform(-0.1, 0.1)    ASCII\_NN.wout.append(round(rw,3))  # Training the neural network ASCII\_NN = BACK\_PROP\_LEARNING(listOfExamples, ASCII\_NN)  # Basic Letter Verification Test for x in range(0, len(listOfExamples)):    letter\_num = 0    maxvalue = 0.0    result = test(listOfExamples[x], ASCII\_NN)    # iterating through the output vector and getting the index of the highest value    for i in range(0, len(result)):        if result[i] > maxvalue:            maxvalue = result[i]            letter\_num = i    print("testing:" + str(Characters[x]) + "\tresult: " + str(Characters[letter\_num]) + "\tvalue: " + str(round(result[letter\_num],3)))  # Bit Flip Test for \_ in range(0, 3):    # each x index corresponds to the english alphabet from A - Z    for x in range(0, len(listOfExamples)):        tolerance = 0        letter\_num = x        flippedSet = set()        # making a deep copy of a letter's example class that will be modified per bit flip        modifiedExample = copy.deepcopy(listOfExamples[x])        display(modifiedExample)        while x == letter\_num:            maxvalue = 0.0            modifiedExample, flippedSet = flip(modifiedExample, flippedSet)            result = test(modifiedExample, ASCII\_NN)            # iterating through the output vector and getting the index of the highest value            for i in range(0, len(result)):                if result[i] > maxvalue:                    maxvalue = result[i]                    letter\_num = i            # checks to see whether the highest index matches the current index being tested                   if x == letter\_num:                tolerance += 1            # prevents an infinite loop, the test can only flip a letter's bits 35 times            if len(flippedSet) == 35:                break         display(modifiedExample)        log = open("test\_data.txt", "a")        log.write("The letter " + Characters[x] + " tolerated " + str(tolerance) + " flips\n")        log.write("The neural network thinks it is the letter " + Characters[letter\_num] + "\n")        log.close()        log = open("bitflip.txt", "a")        log.write(str(tolerance) + "\n")        log.close()        print("The letter " + Characters[x] + " tolerated " + str(tolerance) + " flips")  # Input Test while True:    try:        # Input error checking        inputString = input("Input: ")        while len(inputString) != 35:            print("Not a valid length")            inputString = input("Input: ")        int(inputString)        # Turns a string into a list of characters, then maps each character into an integer value        input\_vector = list(map(int, list(inputString)))        output\_vector = []        letter\_num = 0        maxvalue = 0.0        example = Example(input\_vector, output\_vector)        result = test(example, ASCII\_NN)        # Testing input        for i in range(0, len(result)):            if result[i] > maxvalue:                maxvalue = result[i]                letter\_num = i        print("The neural network thinks your input is the letter " + Characters[letter\_num] + ", " + str(round(result[letter\_num],3)))  display(example)    except ValueError:        print("Not a valid input")        continue | |  | |

Bibliography

1. Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.