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 $1/\sqrt{\#\ of\ input\ neurons}$.

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

```
epoch: 214      error: 0.508     avg: 0.882    time: 0.144
epoch: 215      error: 0.5048    avg: 0.883    time: 0.134
epoch: 216      error: 0.5015    avg: 0.884    time: 0.182
epoch: 217      error: 0.4978    avg: 0.884    time: 0.12
epoch: 218      error: 0.4948    avg: 0.885    time: 0.194
epoch: 219      error: 0.4916    avg: 0.886    time: 0.174
epoch: 220      error: 0.4885    avg: 0.887    time: 0.085
epoch: 221      error: 0.485     avg: 0.888    time: 0.113
epoch: 222      error: 0.482     avg: 0.888    time: 0.173
epoch: 223      error: 0.4789    avg: 0.889    time: 0.176
epoch: 224      error: 0.4758    avg: 0.89     time: 0.138
epoch: 225      error: 0.473     avg: 0.89     time: 0.174
epoch: 226      error: 0.4703    avg: 0.891    time: 0.105
epoch: 227      error: 0.4678    avg: 0.892    time: 0.154
epoch: 228      error: 0.4649    avg: 0.893    time: 0.118
epoch: 229      error: 0.4619    avg: 0.893    time: 0.141
epoch: 230      error: 0.4598    avg: 0.894    time: 0.127
epoch: 231      error: 0.4567    avg: 0.895    time: 0.132
epoch: 232      error: 0.454     avg: 0.895    time: 0.179
epoch: 233      error: 0.4513    avg: 0.896    time: 0.107
epoch: 234      error: 0.4488    avg: 0.896    time: 0.138
epoch: 235      error: 0.4468    avg: 0.897    time: 0.1
epoch: 236      error: 0.4437    avg: 0.898    time: 0.121
epoch: 237      error: 0.4417    avg: 0.898    time: 0.128
epoch: 238      error: 0.4394    avg: 0.899    time: 0.176
epoch: 239      error: 0.4368    avg: 0.899    time: 0.101
epoch: 240      error: 0.4349    avg: 0.9      time: 0.114
```

```
testing:A       result: A       value: 0.906
testing:B       result: B       value: 0.858
testing:C       result: C       value: 0.854
testing:D       result: D       value: 0.915
testing:E       result: E       value: 0.87
testing:F       result: F       value: 0.9
testing:G       result: G       value: 0.874
testing:H       result: H       value: 0.896
testing:I       result: I       value: 0.92
testing:J       result: J       value: 0.94
testing:K       result: K       value: 0.918
testing:L       result: L       value: 0.913
testing:M       result: M       value: 0.913
testing:N       result: N       value: 0.897
testing:O       result: O       value: 0.883
testing:P       result: P       value: 0.883
testing:Q       result: Q       value: 0.905
testing:R       result: R       value: 0.886
testing:S       result: S       value: 0.92
testing:T       result: T       value: 0.922
testing:U       result: U       value: 0.902
testing:V       result: V       value: 0.922
testing:W       result: W       value: 0.904
testing:X       result: X       value: 0.912
testing:Y       result: Y       value: 0.931
testing:Z       result: Z       value: 0.906
```

```
The letter A tolerated 12 flips
The letter B tolerated 10 flips
The letter C tolerated 7 flips
The letter D tolerated 6 flips
The letter E tolerated 9 flips
The letter F tolerated 8 flips
The letter G tolerated 7 flips
The letter H tolerated 7 flips
The letter I tolerated 11 flips
The letter J tolerated 11 flips
The letter K tolerated 11 flips
The letter L tolerated 6 flips
The letter M tolerated 6 flips
The letter N tolerated 8 flips
The letter O tolerated 5 flips
The letter P tolerated 3 flips
The letter Q tolerated 9 flips
The letter R tolerated 8 flips
The letter S tolerated 8 flips
The letter T tolerated 6 flips
The letter U tolerated 8 flips
The letter V tolerated 7 flips
The letter W tolerated 3 flips
The letter X tolerated 9 flips
The letter Y tolerated 10 flips
The letter Z tolerated 7 flips
```

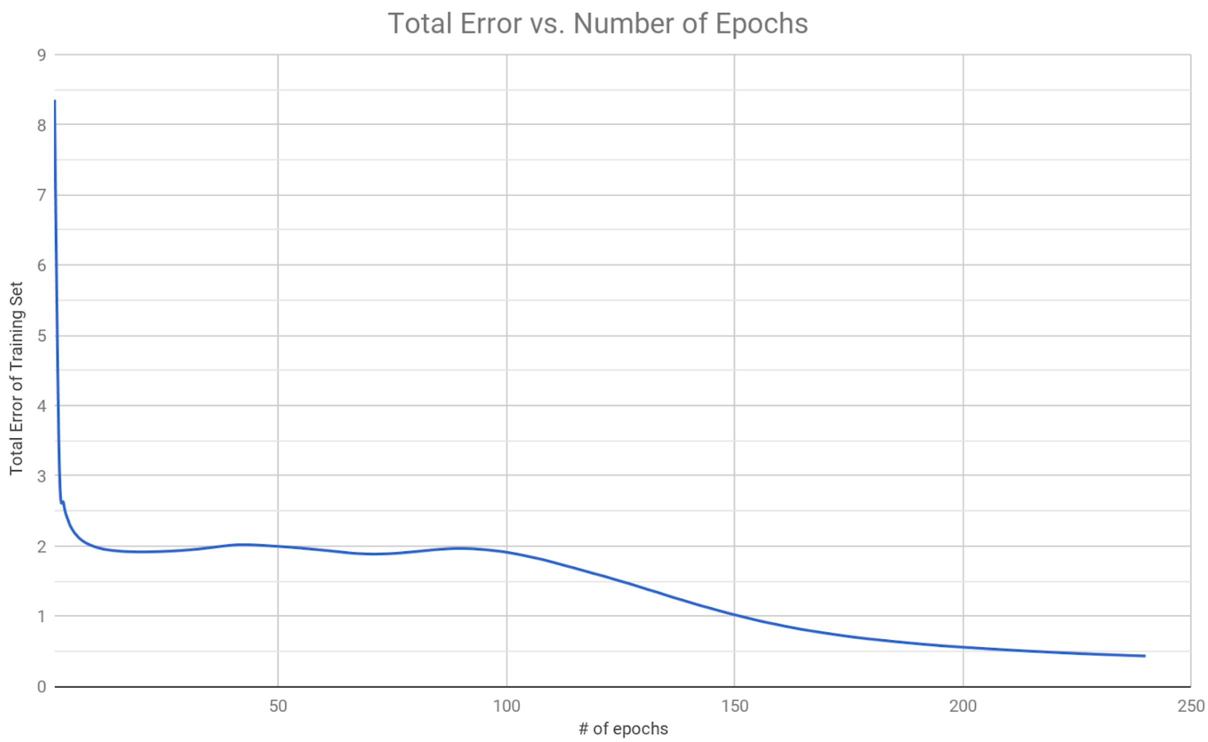
```
Input: 11111100010001001000100100011111110
The neural network thinks your input is the letter A, 0.906
---------------
    1  1  1
 1           1
 1           1
 1           1
 1  1  1  1  1
 1           1
 1           1
---------------

Input:
```

Results and Analysis

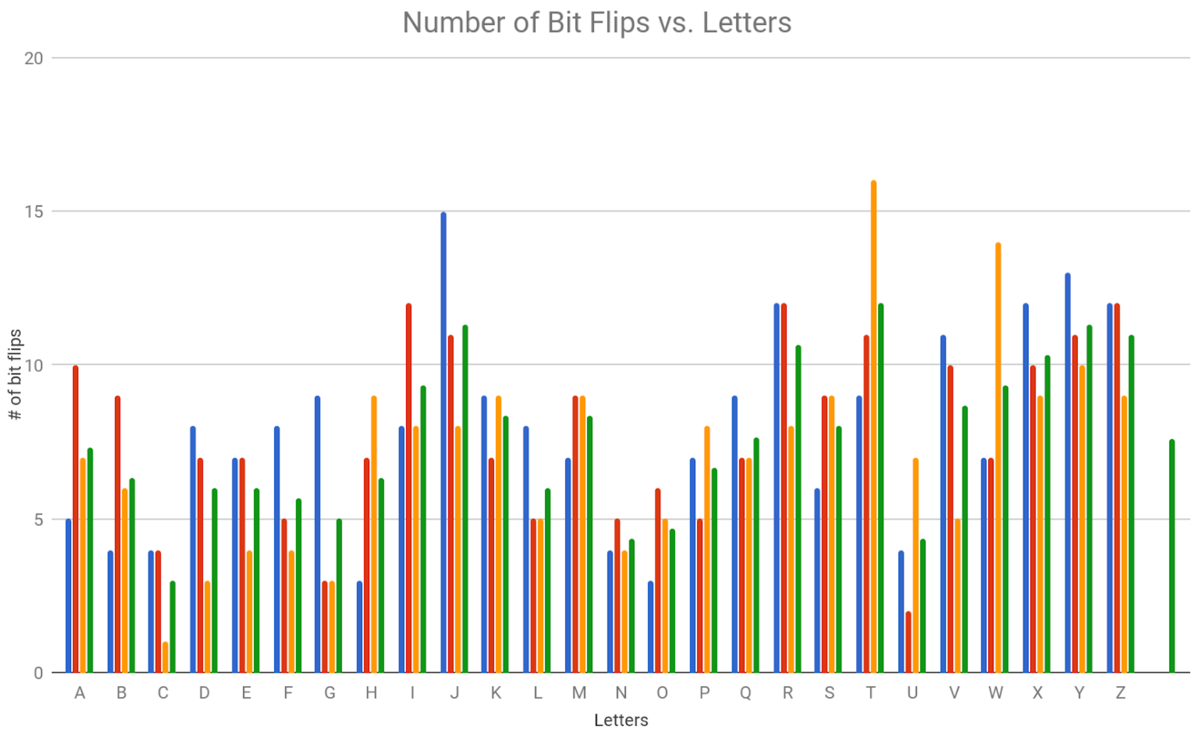
The total error of every epoch was used to graph the learning rate of the neural network. Since each epoch consists of all 26 letters, the calculated total error is the absolute value of the sum of all the error vectors of each letter divided by 26. The values were saved into a text file and graphed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

```python
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)
```

```python
        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.