Artificial Neural Network: An Exploration

Jericho McLeod
CSI-873 Midterm

Data

The dataset used for this project is a sampling of NMIST handwritten digits, ingested as vectors of 785 numbers. The first number indicates the class, and the remaining 784 digits represent the 28x28 matrix of pixel values from 0 to 255. These were scaled to a range of 0 to 1, and then a function was created to visually inspect the data, shown below in figure 1. The dataset contained 60,000 training examples and 10,000 validation examples, all of which were used in this implementation.

Figure 1

Neural Network Functions and Parameters

The Artificial Neural Network created to classify handwritten digits utilized a sigmoid activation function:

$$output = \frac{1}{1 + e^{-z}}$$

Thus the weights were updated using:

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where the updates to weights, with momentum, are calculated by:

$$\Delta w_{ji}(n) = \eta \delta_k x_{ji} + \alpha \Delta w_{ji}(n-1)$$

And the error terms for output and hidden nodes, respectively, are:

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

$$\delta_h = o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k$$

The initial value for momentum (α) is 0.6, and the initial value for the learning rate (η) is 0.3. Weights were initialized randomly between -0.1. and. 0.1, and bias weights were included in the model.

For stopping conditions, several different implementations were tested. It was discovered that waiting for accuracy to converge to some amount tended to be an insufficient measure for this model; either the acceptable range was too broad, and training ended early, or it was unreachable, and accuracy continually shifted by small amounts slightly greater than the stopping condition allowed. Another tested method was moving averages; this however, also led to occasionally early stops. The final selection was for a simpler method; if accuracy decreased twice successively across two iterations, training concluded, and the model reverted to the prior weights. As an additional stopping condition for time concerns, an upper limit of 2^20 iterations of training was implemented.

Neural Network Design Exploration and Results

The network created contained an input for each pixel in the provided arrays, for a total of 784. The hidden layer requirements were for a single layer containing 2, 3, and 4 neurons in each of 3 separate cases. In this implementation, a test was run that used 2 through 10 hidden neurons in order to compare a broader set of results. The output layer chosen was 10, or one for each possible outcome. The network itself is visualized below in figure 2.

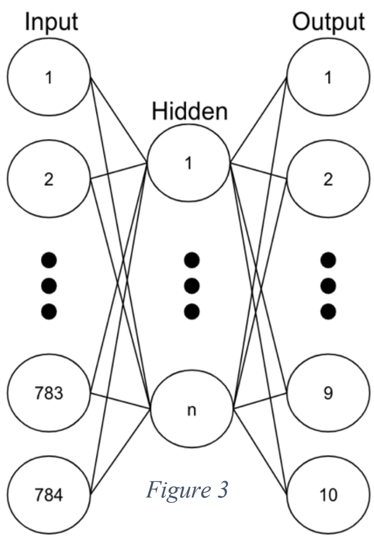

Figure 3

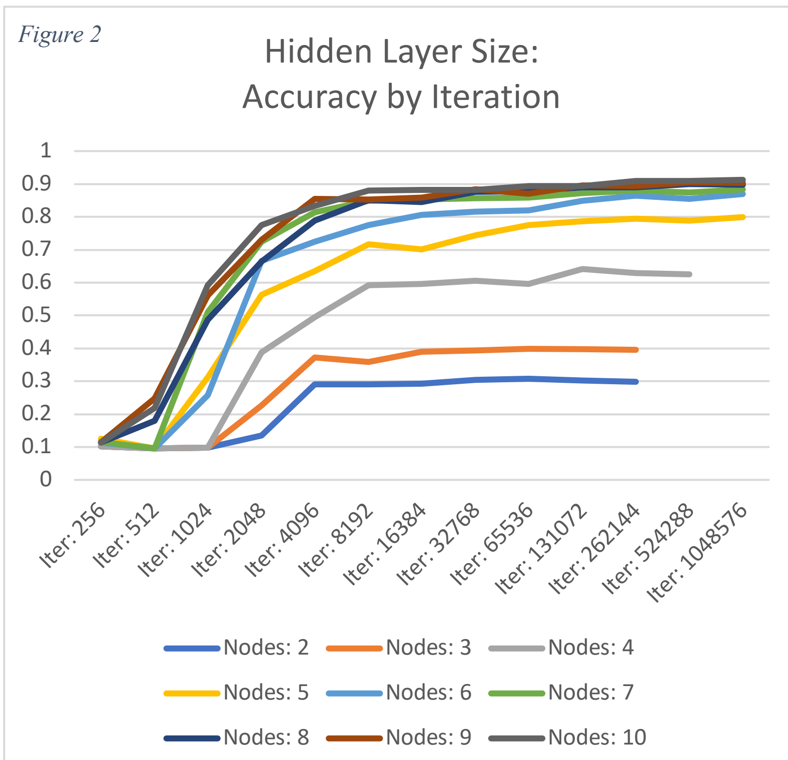

Figure 2

The accuracy obtained for each hidden layer size is show at the opposite in figure 3. The ANNs with 2, 3, and 4 hidden nodes ended training before reaching the specified upper limit, demonstrating that the stopping condition functioned as intended. Time constraints prevented this from being the only possible stopping condition, thus the true limits of accuracy from larger hidden layers remains unknown from this exploration.

Network Weights Exploration

In training the networks, visualizing the weights helps to understand what is being learned. In. figure 4 the training weights for 2, 3, and 4 node hidden layers are shown at 0, 256, 8192, and 32768 iterations of training from left to right, and from top to bottom, 0 to 9, respectively. In these, grey indicates a weigh of 0, while lighter tints indicate negative weights and darker tints indicate positive weights.

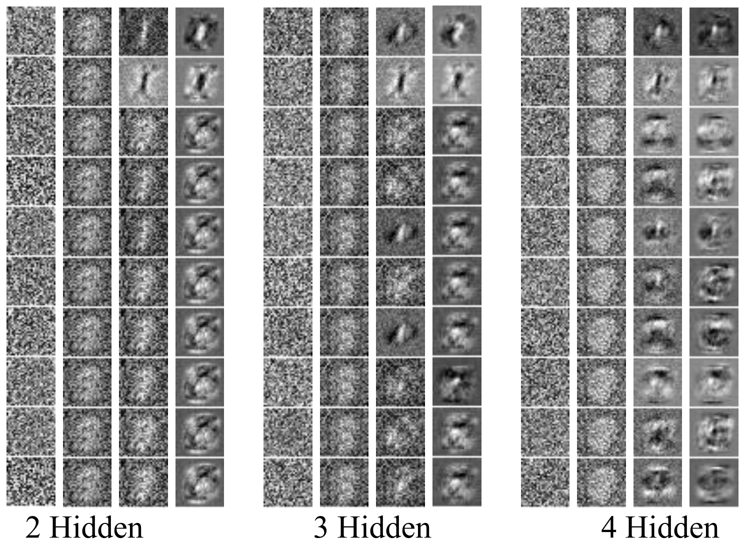

Figure 4

It is easy to see that the initial weights were random, and the limitations of having fewer hidden neurons in terms of how many classifications can be made. Reviewing larger hidden layers shows more easily human-interpretable results. In figure 5 the hidden layers are made up of 6, 8, and 10 nodes, and shown at earlier iterations 0, 256, 2048, and 8192, in figure 5.

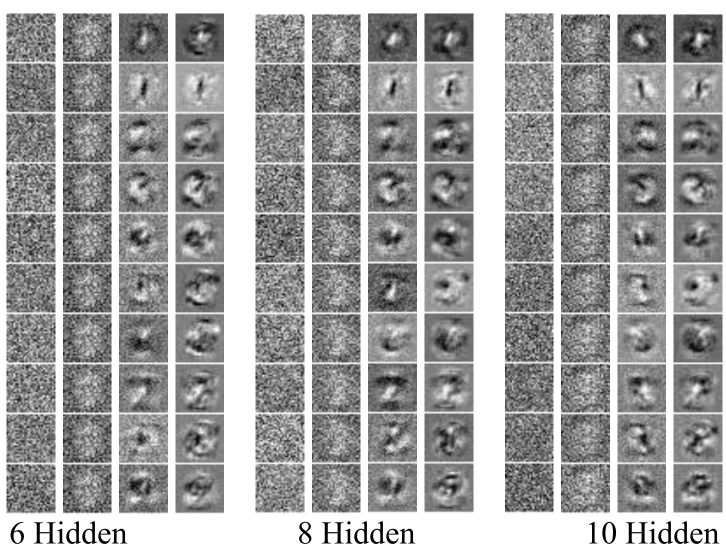

Figure 5

This helps to highlight what visual information may be useful in separating digits. An important distinction to make in reviewing the weights is the expectation is not to see representation of a digit; we are not training a machine to write digits but to recognize them. For this purpose, the location of white space is just as important as black space. Compare, for instance, 3 and 8. The black pixels included in a 3 are also included in an 8; thus, the only way to separate the to is to examine the left side of the digit; if it is black, the digit is an 8, and if white, it is a 3. This is represented in very late training iterations for these digits in figure 6, showing 3 at the left and 8 and the right. Note that digits are considered to be ‘not 3’ if they contain black pixels enclosing the left side, and digits are considered ‘8’ if they contain such pixels.

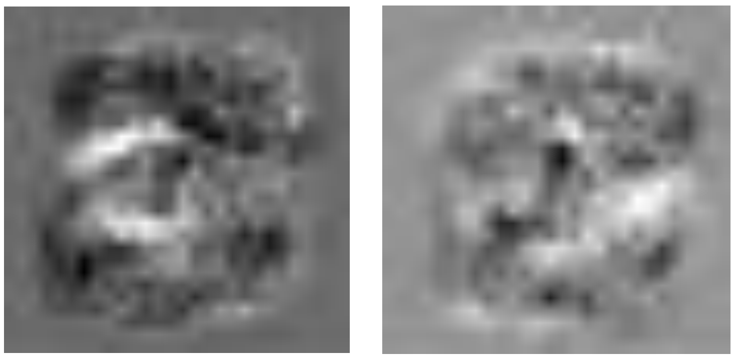

Figure 6

Network Classification Results Exploration

Using two nodes in the hidden layer leads to a relatively low accuracy of approximately 30%, as shown in figure 7. Of this, however, it is noteworthy that two digits are being classified, and essentially all remaining digits are given a single classification that is identical. The true classes represented are “is 0”, “is 1”, and “is not 0 or 1,”; the fact that 3 is successfully classified is incidental.

		Classifed As										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
Actual Class	**0**	913	0	0	67	0	0	0	0	0	0	93.2%
	1	0	1108	0	27	0	0	0	0	0	0	97.6%
	2	3	2	0	1027	0	0	0	0	0	0	0.0%
	3	1	0	0	1009	0	0	0	0	0	0	99.9%
	4	0	2	0	980	0	0	0	0	0	0	0.0%
	5	13	5	0	874	0	0	0	0	0	0	0.0%
	6	8	2	0	948	0	0	0	0	0	0	0.0%
	7	3	9	0	1016	0	0	0	0	0	0	0.0%
	8	5	5	0	964	0	0	0	0	0	0	0.0%
	9	1	5	0	1003	0	0	0	0	0	0	0.0%
		96.4%	97.4%	0.0%	12.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	30.3%

Figure 7

Comparing this to figure 8 we can see that now 3 classes are being well classified when using 3 nodes in the hidden layer, with all remaining objects being classified as a fourth class. This gives approximately 40% accuracy, but one of the classes remains very low in terms of the percentage of correct classifications.

Classifed As

Actual Class	0	1	2	3	4	5	6	7	8	9	
0	938	0	0	41	0	0	0	1	0	0	95.7%
1	0	1099	0	35	0	0	0	1	0	0	96.8%
2	8	4	0	1004	0	0	0	16	0	0	0.0%
3	3	0	0	995	0	0	0	12	0	0	98.5%
4	0	1	0	979	0	0	0	2	0	0	0.0%
5	22	4	0	854	0	0	0	12	0	0	0.0%
6	13	3	0	942	0	0	0	0	0	0	0.0%
7	2	10	0	81	0	0	0	935	0	0	91.0%
8	10	5	0	955	0	0	0	4	0	0	0.0%
9	2	5	0	986	0	0	0	16	0	0	0.0%
	94.0%	97.2%	0.0%	14.5%	0.0%	0.0%	0.0%	93.6%	0.0%	0.0%	39.7%

Figure 8

Once the number of nodes in the hidden layer reaches 4, shown in figure 9, we begin to see a greater rate of classifications than nodes; the model is now attempting to classify 7 digits, with remaining outputs falling to one of the presently unused classes.

Classifed As

Actual Class	0	1	2	3	4	5	6	7	8	9	
0	857	0	7	1	9	0	3	103	0	0	87.4%
1	0	1079	1	6	0	0	42	5	2	0	95.1%
2	29	23	828	19	8	0	77	45	3	0	80.2%
3	7	21	14	831	2	0	10	122	3	0	82.3%
4	2	1	3	3	889	0	40	44	0	0	90.5%
5	13	9	4	32	28	0	39	761	6	0	0.0%
6	5	2	12	0	36	0	871	30	2	0	90.9%
7	6	17	11	22	9	0	32	915	16	0	89.0%
8	0	6	0	19	20	0	125	785	19	0	2.0%
9	3	1	1	4	64	0	6	926	4	0	0.0%
	93.0%	93.1%	94.0%	88.7%	83.5%	0.0%	70.0%	24.5%	34.5%	0.0%	62.9%

Figure 9

And at hidden layer sizes of 5 or greater we begin to see all classifications being utilized, as shown in figure 10. It is possible that training for a greater length of time would improve the results such that the maximum information that could be contained in the hidden layer, 2^n where n is the number of nodes, could be reached; however, the marginal return for training times is not ideal. Training 1,000,000 iterations with a smaller number of nodes takes much longer than adding hidden neurons and training for a much shorter time period to reach the same or better results.

Classifed As

Actual Class	0	1	2	3	4	5	6	7	8	9	
0	908	1	0	10	7	31	15	1	7	0	92.7%
1	0	1114	1	7	0	2	1	4	6	0	98.1%
2	19	73	694	137	17	10	43	3	33	3	67.2%
3	5	69	33	776	0	100	4	18	2	3	76.8%
4	0	3	6	0	806	0	12	15	5	135	82.1%
5	106	30	7	44	14	593	4	18	71	5	66.5%
6	26	7	16	7	51	3	816	4	27	1	85.2%
7	9	45	11	17	8	1	16	893	0	28	86.9%
8	15	269	19	10	19	36	6	7	578	15	59.3%
9	15	16	0	3	95	17	3	35	11	814	80.7%
	82.3%	68.5%	88.2%	76.8%	79.3%	74.8%	88.7%	89.5%	78.1%	81.1%	79.9%

Figure 10

Examining the results of a neural network with 10 nodes in the hidden layer shows much better results overall, as seen in figure 11. It is possible that neural networks perform best when the hidden layer size is greater than or equal to the output layer size; however, the scope of that question is beyond this exploration.

Classifed As

Actual Class	0	1	2	3	4	5	6	7	8	9	
0	948	0	4	3	7	5	8	3	2	0	96.7%
1	0	1105	2	6	0	2	6	2	12	0	97.4%
2	12	13	902	21	8	7	29	9	30	1	87.4%
3	4	4	22	877	0	41	4	9	48	1	86.8%
4	4	0	3	1	908	0	21	2	13	30	92.5%
5	16	5	5	33	2	777	16	6	28	4	87.1%
6	24	1	9	1	7	9	904	0	3	0	94.4%
7	0	15	27	16	7	1	0	936	2	24	91.1%
8	9	8	5	27	4	31	23	6	856	5	87.9%
9	7	3	2	9	23	5	4	6	36	914	90.6%
	92.6%	95.8%	91.9%	88.2%	94.0%	88.5%	89.1%	95.6%	83.1%	93.4%	91.3%

Figure 11

Misclassification Exploration

Visually comparing data that was accurately classified with misclassified data helps to understand misclassifications. For this purpose, the 10-hidden-node network was used. In the two arrays below, figures 12 and 13, one can see that the digits that are misclassified are atypical, yet still recognizable to humans for the most part. In contrast to these, correctly classified digits tend to be written in more typical forms. This highlights that the Artificial Neural Network is classifying with limited features and is not able to easily identify digits with additional or missing serifs, altered angles, or extra pixels entered as noise.

Incorrectly Classified:

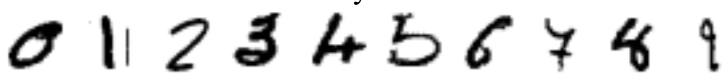

Figure 12

Correctly Classified:

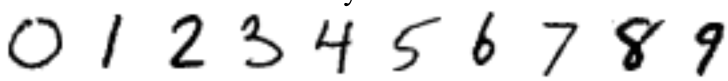

Figure 13

To create a more model that is more resilient to misclassifications, one may wish to consider deleting blocks of pixels from training data, apply transformations to stretch or skew images, or to add additional blocks of pixels. Doing this for the entire data set, while also retaining the original, and training on the augmented collection of training examples, would force the network to expand the features being used to classify digits, thus potentially increasing performance. However, this would also increase training times, and may be of limited benefit without also expanding the size of the hidden layer and even the depth of the network via additional hidden layers.

Code Appendix

Notes on running this algorithm:

1) The input data must be located in the same director as this file, and must not be encrypted or compressed.
2) The algorithm will provide 3 sets of output, with output files located in the directory from which the script is run:
    a. Printing summary results in the console
    b. Saving a .csv copy of the results summary
    c. Saving an image of the weights at each checkpoint for each hidden node quantity
3) Libraries required:
    a. CSV
    b. Random
    c. Math
    d. Decimal
    e. Copy
    f. Numpy
    g. PIL

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Oct  5 15:41:29 2019

@author: jmcleod

CSI-873: Computational Learning

This is an Artificial Neural Network with a single hidden layer of some
specified size. I used it to classify NMIST data for handwritten digits, thus
this file also contains image generation functions to convert the numeric
arrays back into graphics for human review.

The activation function is sigmoid: 1 / 1  + e ^ -z
"""

import csv,random,math,decimal,copy
import numpy as np
from PIL import Image
decimal.getcontext().prec = 100


def data_import(file):
    '''
    This function imports the data from a particular file
    and returns an array of arrays
    '''
    data = []
    with open(file, 'r') as csvfile:
        csv_r = csv.reader(csvfile,delimiter=' ')
        for row in csv_r:
            row_nums = []
            for i in range(len(row)):
                try:
                    val = float(row[i])
                    if i > 0:
```

```
                        val = round(val/255,4)
                        # The above line scales the data imported
                    row_nums.append(val)
                except:
                    print('ERROR on import: non-numerical data:',row[i])
                    break
            data.append(row_nums)
    return(data)

def data_import_loop(string,denom):
    '''This function loops the data import across all files of the chosen type,
    which is specified by the string argument passed to the function.
    It then uses the first value in the set to add the imported arrays
    to the correct dictionary key, created with values 0-9.
    The resulting dictionary is returned.
    '''
    files = []
    data_dict = {}
    for i in range(10):
        file_name = string+str(i)+'.txt'
        files.append(file_name)
        data_dict[i]=[]
    for i in files:
        data = data_import(i)
        for j in range(len(data)):
            if j%denom==0: # SUBSET data
                data_dict[data[j][0]].append(data[j][1:])
    return(data_dict)

def create_image_data(char_matrix):
    '''
    This function outputs a human-viewable copy of an input from matrix form
    '''
    data = np.zeros( (len(char_matrix),len(char_matrix[0]),3), dtype=np.uint8 )
    for row in range(len(char_matrix)):
        for col in range(len(char_matrix[row])):
            val = 255 - char_matrix[col][row]
            data[row,col] = [val,val,val]
    return(data)

def create_large_image(data_dict):
    '''This function creates an NxN image of 10 examples of 10 classes'''
    shortest = 1000000
    for k,v in data_dict.items():
        if len(v) < shortest:
            shortest = len(v)
    big_matrix_data = []
    for m in range(10):
        medium_matrix_data = []
        for i in range(28):
            medium_matrix_data.append([])
        for i in range(10):
            random_num = random.randint(0,shortest-1)
            array = data_dict[m][random_num]
            for j in range(len(array)):
                medium_matrix_data[j%28].append((array[j]*255))
        for i in medium_matrix_data:
            big_matrix_data.append(i)
    big_image = create_image_data(big_matrix_data)
    image = Image.fromarray(big_image)
    image.show()

def randomize_data_arrays(data_dict):
```

```
    ''' This is a function to randomize the order of training and test data'''
    data_array = []
    data_result = []
    for k,v in data_dict.items():
        for i in v:
            data_result.append(k)
            data_array.append(i)
    random_index = []
    for i in range(len(data_array)):
        random_index.append(random.random())
    random_index_copy = copy.deepcopy(random_index)
    rand_data_array = []
    rand_data_result = []
    for i in range(len(random_index)):
        min_val = min(random_index_copy)
        random_index_copy.pop(random_index_copy.index(min_val))
        index_val = random_index.index(min_val)
        rand_data_array.append(data_array[index_val])
        rand_data_result.append(data_result[index_val])
    data_array = rand_data_array
    data_result = rand_data_result
    return(data_array,data_result)

class neuron:
    def __init__(self,input_count,starting_weight,learn_rate):
        self.weights = [starting_weight]*(input_count+1)
        self.delta_weights = [0]*(input_count+1)
        for i in range(input_count+1):
            rando = random.uniform(-starting_weight,starting_weight)
            self.weights[i] = rando
        self.learn_rate=learn_rate
        self.output = 0

class output_neuron(neuron):
    def feed_forward(self,input_array):
        x = self.weights[0]
        for i in range(len(input_array)):
            x += float(input_array[i])*float(self.weights[i+1])
        x_out =
decimal.Decimal(1)/(decimal.Decimal(1)+(decimal.Decimal(math.e)**(decimal.Decimal(-
x)))) # Sigmoid output
        x_out = float(round(x_out,16))
        self.output = x_out
        return(x_out)
    def back_prop(self,t_o,inputs,momentum):
        error = self.output * (1 - self.output) * (t_o - self.output)
        for i in range(len(self.weights)):
            try: xji = inputs[i-1]
            except: xji = 1
            prior_weight_delta = self.delta_weights[i]
            self.weights[i] = (self.learn_rate * error * xji) + self.weights[i] +
(momentum * prior_weight_delta)
            self.delta_weights[i] = self.learn_rate * error * xji

class hidden_neuron(neuron):
    def feed_forward(self,input_array):
        x = self.weights[0]
        for i in range(len(input_array)):
            x += float(input_array[i])*float(self.weights[i+1])
        x_out =
decimal.Decimal(1)/(decimal.Decimal(1)+(decimal.Decimal(math.e)**(decimal.Decimal(-
x)))) # Sigmoid output
        x_out = float(round(x_out,16))
```

```
        self.output = x_out
        return(x_out)
    def back_prop(self,w_e_term,inputs,momentum):
        error = w_e_term * self.output*(1-self.output)
        for i in range(len(self.weights)):
            try: xji = inputs[i-1]
            except: xji = 1
            prior_weight_delta = self.delta_weights[i]
            self.weights[i]= (self.learn_rate * error * xji) + self.weights[i] +
(prior_weight_delta * momentum)
            self.delta_weights[i] = self.learn_rate * error * xji

class neural_network:
    def __init__(self,dataset,classes,hidden_neurons,output_neurons,\
                 starting_weight=0.1,learn_rate=0.3,momentum = 0.6):
        self.inputs =  len(dataset[0])
        self.dataset = dataset
        self.classes = classes
        self.starting_weight = starting_weight
        self.learn_rate = learn_rate
        self.momentum = momentum
        self.hidden_layer = []
        self.output_layer  = []
        self.output_errors = []
        for i in range(hidden_neurons):
            self.hidden_layer.append(hidden_neuron(self.inputs,\
                                                   self.starting_weight,\
                                                   self.learn_rate))
        for i in range(output_neurons):
            self.output_layer.append(output_neuron(len(self.hidden_layer),\
                                                   self.starting_weight,\
                                                   self.learn_rate))
        self.hidden_x = []
        self.output_x = []

    def feed_forward(self,epoch):
        data_instance = self.dataset[(epoch % len(self.dataset))]
        self.hidden_x = []
        for n in self.hidden_layer:
            self.hidden_x.append(n.feed_forward(data_instance))
        self.output_x = []
        for n in self.output_layer:
            self.output_x.append(n.feed_forward(self.hidden_x))

    def back_prop(self,iteration):
        self.output_errors = []
        hidden_errors = []
        target_class = self.classes[(iteration%len(self.dataset))]
        target_outputs = [0.01]*len(self.output_layer)
        delta_weights = []
        for i in range(len(self.output_layer)):
            if i==target_class:
                target_outputs[i] +=0.98
        for n in range(len(self.output_layer)):
            neuron = self.output_layer[n]
            self.output_errors.append(neuron.output * (1  - neuron.output) * \
                                     (target_outputs[n] - neuron.output))
        for n in range(len(self.hidden_layer)):
            neuron = self.hidden_layer[n]
            output = neuron.output
            pre_error = output * (1-output)
            wk = 0
            for n2 in range(len(self.output_layer)):
```

```
                o_neuron = self.output_layer[n2]
                wk += (o_neuron.weights[n+1] * self.output_errors[n2])
            hidden_errors.append(wk * pre_error)
        for n in range(len(self.output_layer)):
            neuron = self.output_layer[n]
            for w in range(len(neuron.weights)):
                try: xji = self.hidden_x[w-1]
                except: xji = 1
                delta_w = neuron.learn_rate * self.output_errors[n] * xji
                delta_weights.append(delta_w)
                neuron.weights[w] += (delta_w + self.momentum *
neuron.delta_weights[w])
                neuron.delta_weights[w] = delta_w
        for n in range(len(self.hidden_layer)):
            neuron =  self.hidden_layer[n]
            for w in range(len(neuron.weights)):
                try: xji = self.dataset[(iteration%len(self.dataset))][w-1]
                except: xji = 1
                delta_w = neuron.learn_rate * hidden_errors[n] * xji
#self.output_errors[n]
                delta_weights.append(delta_w)
                neuron.weights[w] += (delta_w + self.momentum *
neuron.delta_weights[w])
                neuron.delta_weights[w] = delta_w
        return(delta_weights)

    def classify(self,array):
        self.hidden_x = []
        for n in self.hidden_layer:
            self.hidden_x.append(n.feed_forward(array))
        self.output_x = []
        for n in self.output_layer:
            self.output_x.append(n.feed_forward(self.hidden_x))
        classification = 0
        value = 0
        for i in range(len(self.output_x)):
            if self.output_x[i]>=value:
                classification = i
                value = self.output_x[i]
        return(classification)

def measure_model(test_set,test_answers,model):
    '''This function determines the accuracy of a model by classifying the
    test data'''
    total = 0
    correct  = 0
    for n in range(len(test_answers)):
        rando = random.random()
        if rando > 0: #set to higher value to subset data
            classification = model.classify(test_set[n])
            actual_class = test_answers[n]
            if classification == actual_class:
                correct+=1
            total+=1
    return(correct/total)

def conf_matrix_shell():
    '''This function just returns an empty NxN matrix'''
    matrix = []
    for i in range(10):
        row=[]
        for j in range(10):
            row.append(0)
```

```
        matrix.append(row)
    return(matrix)

def get_conf_matrix(test_set,test_answers,model):
    '''This function uses the NxN matrix created above and adds
    observed classifications'''
    matrix = conf_matrix_shell()
    for n in range(len(test_answers)):
        rando = random.random()
        if rando > 0: #adjust to subset data
            classification = model.classify(test_set[n])
            actual_class = test_answers[n]
            matrix[actual_class][classification]+=1
    return(matrix)

def get_weights_image_vector(model):
    hidden_weights = []
    output_weights = []
    pixel_weights = {}
    for i in model.hidden_layer:
        hidden_weights.append(i.weights[1:])
    for i in model.output_layer:
        output_weights.append(i.weights[1:])

    for out in range(len(output_weights)):
        vector = []
        for i in range(len(hidden_weights[0])):
            vals,val = [],0
            for h in hidden_weights:
                vals.append(h[i])

            for o in range(len(output_weights[out])): #
                vals[o] = vals[o] * output_weights[out][o] #
            for v in vals:
                val += v
            vector.append(val)
        pixel_weights[out] = vector
    return(pixel_weights)

def min_max_array(array):
    out = []
    for i in array:
        v = (i-min(array)) / (max(array)-min(array))
        out.append(v)
    return(out)


def alt_min_max(array):
    out,temp = [],[]
    for i in array:
        if i>=0:
            temp.append(i)
        else:
            temp.append(i*-1)
    for i in temp:
        v = (i-min(temp)) / (max(temp)-min(temp))
        out.append(v)
    return(out)




def create_small_image_data(char_matrix):
```

```
    ''' Create an image of a specific input
    Useful after the above classification command
    in order to see the image being classified'''
    data = np.zeros( (len(char_matrix),len(char_matrix[0]),3), dtype=np.uint8 )
    for row in range(len(char_matrix)):
        for col in range(len(char_matrix[row])):
            #print(col,row,len(char_matrix),len(char_matrix[0]))
            val = 255- (255 * char_matrix[row][col])
            data[row,col] = [val,val,val]
    return(data)


def create_image(data_array,name):
    matrix = []
    for i in range(int(len(data_array)/28)):
        row = data_array[i*28:i*28+28]
        matrix.append(row)
    image_matrix = create_small_image_data(matrix)
    image = Image.fromarray(image_matrix)
    filename = str(name)+'.jpg'
    image.save(filename)



def create_single_array(model,i):
    ''' Creates the array for weights images'''
    pixel_weights = get_weights_image_vector(model)
    blank_row = [0]*28
    temp_vec1 = min_max_array(pixel_weights[0])
    temp_vec2 = min_max_array(pixel_weights[1])
    temp_vec3 = min_max_array(pixel_weights[2])
    temp_vec4 = min_max_array(pixel_weights[3])
    temp_vec5 = min_max_array(pixel_weights[4])
    temp_vec6 = min_max_array(pixel_weights[5])
    temp_vec7 = min_max_array(pixel_weights[6])
    temp_vec8 = min_max_array(pixel_weights[7])
    temp_vec9 = min_max_array(pixel_weights[8])
    temp_vec10 = min_max_array(pixel_weights[9])
    temp_array = temp_vec1+blank_row+\
                 temp_vec2+blank_row+\
                 temp_vec3+blank_row+\
                 temp_vec4+blank_row+\
                 temp_vec5+blank_row+\
                 temp_vec6+blank_row+\
                 temp_vec7+blank_row+\
                 temp_vec8+blank_row+\
                 temp_vec9+blank_row+\
                 temp_vec10+blank_row
    create_image(temp_array,i)


def main():
    '''Data Import'''
    denom = 1
    data_dict = data_import_loop('train',denom)
    denom = 1
    test_dict = data_import_loop('test',denom)
    '''Create sample image of data'''
    create_large_image(data_dict)
    '''Randomize the order of the data'''
    data_array,data_result = randomize_data_arrays(data_dict)
    test_array,test_result = randomize_data_arrays(test_dict)
    file_output = []
```

```
    for i in range(2,11):
        prior_ann = 0
        prior_accuracy = 0
        prior_accs = []
        stop = 0
        powers = 8
        data_instance = 0
        ann = neural_network(data_array,data_result,i,10)
        name = str(i)+'_nodes_'+str(data_instance)+'_iters'
        create_single_array(ann,name)
        while stop < 1:
            for n in range(2**powers-data_instance):
                ann.feed_forward(data_instance)
                ann.back_prop(data_instance)
                data_instance+=1
            name = str(i)+'_nodes_'+str(data_instance)+'_iters'
            create_single_array(ann,name)
            result = measure_model(test_array,test_result,ann)
            prior_accs.append(result)
            print('Nodes: %d  Iterations: %d   Accuracy:
%4f'%(i,data_instance,result))
            file_output.append(['Nodes: %d  Iterations: %d   Accuracy:
%4f'%(i,data_instance,result)])
            powers+=1
            delta_acc = result - prior_accuracy
            print(result,prior_accuracy,delta_acc)
            if delta_acc < 0: delta_acc = delta_acc * -1
            if delta_acc < .005 and data_instance > 8200: #stopping condition on drop
in accuracy
                stop+=1
            elif len(prior_accs) > 2:
                if prior_accs[-1]<prior_accs[-2] and  prior_accs[-2] < prior_accs[-3]:
#stopping condition for 10% decrease in accuracy with rewind to prior ANN
                    stop+=1
                    ann = prior_ann
            else:
                prior_accuracy = result
            prior_ann = copy.deepcopy(ann)
            if powers > 15: #upper bound on iterations to train
                stop+=1
        confusion_matrix = get_conf_matrix(test_array,test_result,ann)
        print()
        print('Hidden Nodes:',i)
        print('Accuracy:',result)
        print('Confusion Matrix:')
        for i in confusion_matrix:
            print(i)
        print()
        file_output.append([])
        file_output.append(['Hidden Nodes:',i])
        file_output.append(['Accuracy:',result])
        file_output.append(['Confusion Matrix:'])
        for i in confusion_matrix:
            file_output.append(i)
        file_output.append([])

    with open('output_filename.csv', mode='w') as csvfile:
        csv_r = csv.writer(csvfile,delimiter=',')
        for row in file_output:
            csv_r.writerow(row)

if __name__ == '__main__':
    main()
```