**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.

![A close up of a sign

Description automatically generated]()

Figure

## **Neural Network Functions and Parameters**

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

Thus the weights were updated using:

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

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

The initial value for momentum () is 0.6, and the initial value for the learning rate () 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.

A picture containing text

Description automatically generated

Figure

Figure

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.

           

2 Hidden 3 Hidden 4 Hidden

Figure

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.

           

6 Hidden 8 Hidden 10 Hidden

Figure

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

## **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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Classified 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

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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Classified As | | | | | | | | | |  |
|  |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |  |
| Actual Class | **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

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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Classified As | | | | | | | | | |  |
|  |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |  |
| Actual Class | **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

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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Classified As | | | | | | | | | |  |
|  |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |  |
| Actual Class | **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

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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Classified As | | | | | | | | | |  |
|  |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |  |
| Actual Class | **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

## **Confidence Intervals**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **2 N** | **3 N** | **4 N** | **5 N** | **6 N** | **7 N** | **8 N** | **9 N** | **10 N** |
| **Upper 95% CI** | 0.307 | 0.406 | 0.635 | 0.807 | 0.876 | 0.890 | 0.904 | 0.909 | 0.918 |
| **Mean** | 0.298 | 0.397 | 0.625 | 0.799 | 0.870 | 0.884 | 0.898 | 0.903 | 0.913 |
| **Lower 95% CI** | 0.289 | 0.387 | 0.616 | 0.791 | 0.863 | 0.877 | 0.892 | 0.898 | 0.907 |

Given that the test dataset used was 10,000 instances, the confidence intervals for the tested networks is relatively small. There is notably little overlap between the smaller hidden layers, while the larger hidden layers begin to overlap for the 95% confidence intervals for overall accuracy.

## **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

Correctly Classified:



Figure

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:
   1. Printing summary results in the console
   2. Saving a .csv copy of the results summary
   3. Saving an image of the weights at each checkpoint for each hidden node quantity
3. Libraries required:
   1. CSV
   2. Random
   3. Math
   4. Decimal
   5. Copy
   6. Numpy
   7. 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()