**1. Introduction**

In computation, linear or limited variable problems often have straight-forward and low-cost approximate solutions. These problems often entail an algorithmic approach in which the scope of the solution is already well-defined. However, in numerous machine learning problems, the scope of the solution is not as well-defined, and the structural algorithmic approach must be replaced by an adaptable one instead. In this way, computers would be able to develop solutions that the developer hasn’t already had the foresight to predict.

In particular, humans are generally both efficient and accurate in their approach to multivariate problems. For example, after having seen many examples of what a “tree” would look like in their lives, an individual would be able to observe the numerous attributes of any object and decide whether or not to classify it as a “tree.” However, in contrast to the power of computation, human learning and evaluation speeds of new and foreign problems are relatively slow.

An Artificial Neural Network (ANN) is a machine learning model loosely based on the biological structure of the brain, aimed to emulate and expedite the learning process observed in humans and other animals. In the past, ANNs have shown considerable success in solving other multivariable learning problems, including stock market predictions, computer vision, and finding approximate solutions to the Travelling Salesman Problem (TSP). Neural networks have also shown success in developing intelligences that can perform image classification and have shown particular success in optical character recognition software. In general, the layout of a neural network is specific to the problem at hand. However, the design of a neural network is algorithmic in itself and can be made to be extensible across a discrete classification of problem sets.

In particular, ANNs success in supervised image classification offers a promising field of exploration. One potentially implementable form of this is with Optical Digit Recognition of handwritten characters in order to convert handwritten text into computer-type. This would allow computers to compile handwritten text into easy to access and searchable documents. Furthermore, ANNs would help remove human error, learning cost, and lack of efficiency in higher-caliber tasks. In particular, neural networks could help diagnose and analyze medical images including but not limited to biopsies, x-rays, MRIs, and CAT scans. With larger and larger sets of data, not only would this make diagnoses more accurate and efficient, but it also considerably lower the cost of diagnosis and make it more affordable. According to statistics from Cancer Research UK, more than 90% of women diagnosed with breast cancer at the earliest stage survive their disease for at least 5 years compared to around 15% for women diagnosed with the most advanced stage of disease. This could potentially save lives if terminable diseases such as cancer are able to be detected early.

**1.1 Goal**

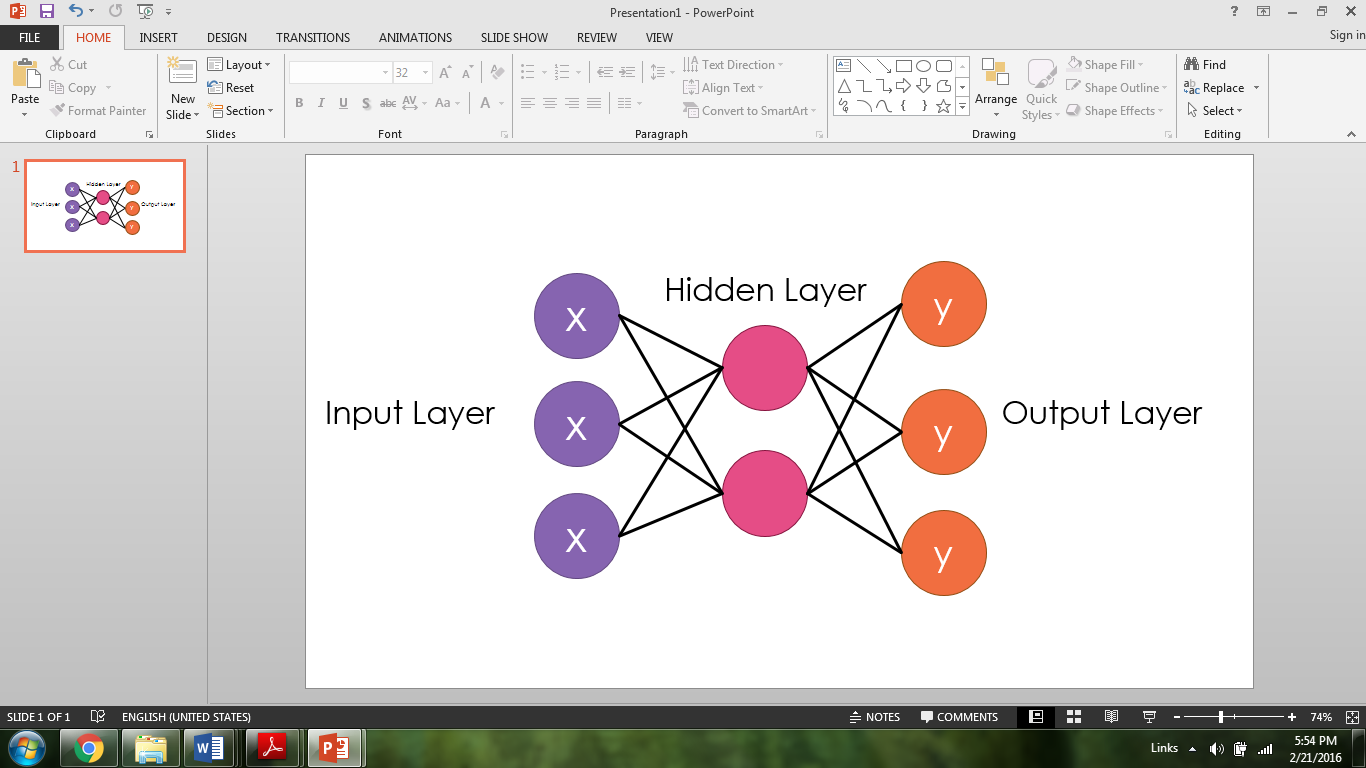
The goal of this research was to develop a general-purpose, extensible artificial neural network for the supervised analysis and classification of images and to apply this ANN for Optical Digit Classification and categorical diagnosis of fine needle aspirates of breast cancer tumors.

**2. Methods**

This project was programmed with an object-oriented approach using the Python programming language using the pygame, scipy, numpy, and pillow libraries.

**2.1 Developing an Extensible ANN**

**The neural network framework.** The first step of this project is to develop the extensible neural network, able to be applied to any number of classification problems. The canonical three-layered neural network was used (see **Figure 1** below), containing the input layer, hidden layer, and output layer. In this particular approach, only one hidden layer was implemented, as prior methods showed that deeper or more convoluted neural networks did not entirely benefit neural networks to a great extent in proportion to computational cost. The number of neurons in the input and output layers were constants dependent on the nature of the training data. The number of neurons in the hidden layer is also dependent on the training data, but is determined using iterative testing discussed later on. The value holding nodes in each layer and the connections between them will be referred to as neurons and synapses respectively in correlation to the terminology of brain structure.



**Figure 1. General framework for the three-layered neural network**

This is the constructor used to initiate Neural\_Network class as previously described:

**class** **Neural\_Network(**object**):**

#Intialize neural network object (requires length of square image)

**def** \_\_init\_\_ **(**self**,** iLayer**=**2**,** oLayer**=**10**,** hLayer**=**10**,** Lambda**=**0**):**

self**.**inputLayerSize**=** iLayer

self**.**outputLayerSize **=** oLayer

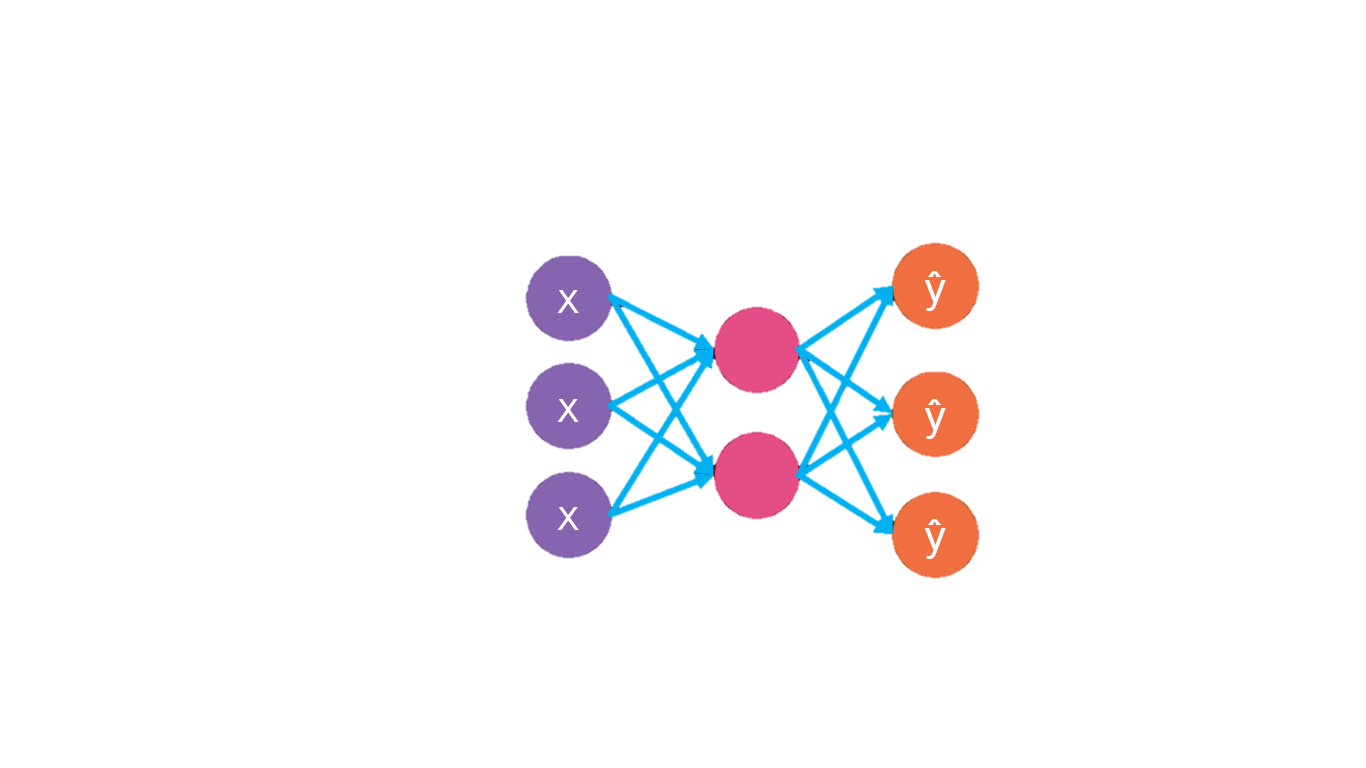
self**.**hiddenLayerSize **=** hLayer

self**.**W1 **=** np**.**random**.**randn**(**self**.**inputLayerSize**,** self**.**hiddenLayerSize**)**

self**.**W2 **=** np**.**random**.**randn**(**self**.**hiddenLayerSize**,** self**.**outputLayerSize**)**

self**.**Lambda **=** Lambda

**Forward propagation.** ANNs operate on the basis of giving the inputs varying weights or biases in determining the nature of the outputs. This process occurs as illustrated in **Figure 2** below.



Weights (W1)

Weights (W2)

Activation (f(z1))

Activation (f(z2))

**Figure 2. Forward propagation illustrated in the neural layout**

Data begins in the input layer, where it is inputted. From here, a weight is applied to each input and the resulting values are summed at each neuron of the hidden layer. As it is seen in the above constructor, the number of weights between two layers is dependent on the number of neurons in each of the two endpoint layers because each neuron in the initial layer is connected to each neuron in the subsequent layer. Because these weighted inputs are being summed at each neuron of the hidden layer, it is useful to treat this operation as a matrix multiplication in order to simplify notation. At each subsequent neuron (after the input layer), an activation function is applied to the received value, which becomes the new input. This process is repeated until values reach the output layer and there are no more synapses to traverse. In the case of a simple ANN, as it is in the case of this implementation, this process is only repeated two fold, but an algorithmic implementation can easily be extended for deeper neural networks.

The general forward propagation model can be defined by the following series of equations, in which X is the input matrix, W1 and W2 are the weight matrices, f(z) is the activation function (which will be later discussed), and is the predicted output:

|  |  |
| --- | --- |
|  | [1] |
|  | [2] |
|  | [3] |
|  | [4] |
|  | [5] |

Equation 5 above is the composed version of its predecessors. These series of equations was used to implement the forward propagation method in code:

**def** forward**(**self**,** x**):**

self**.**z2 **=** np**.**dot**(**x**,** self**.**W1**)**

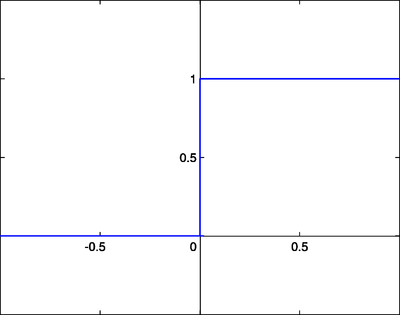
self**.**a2 **=** self**.**sigmoid**(**self**.**z2**)**

self**.**z3 **=** np**.**dot**(**self**.**a2**,** self**.**W2**)**

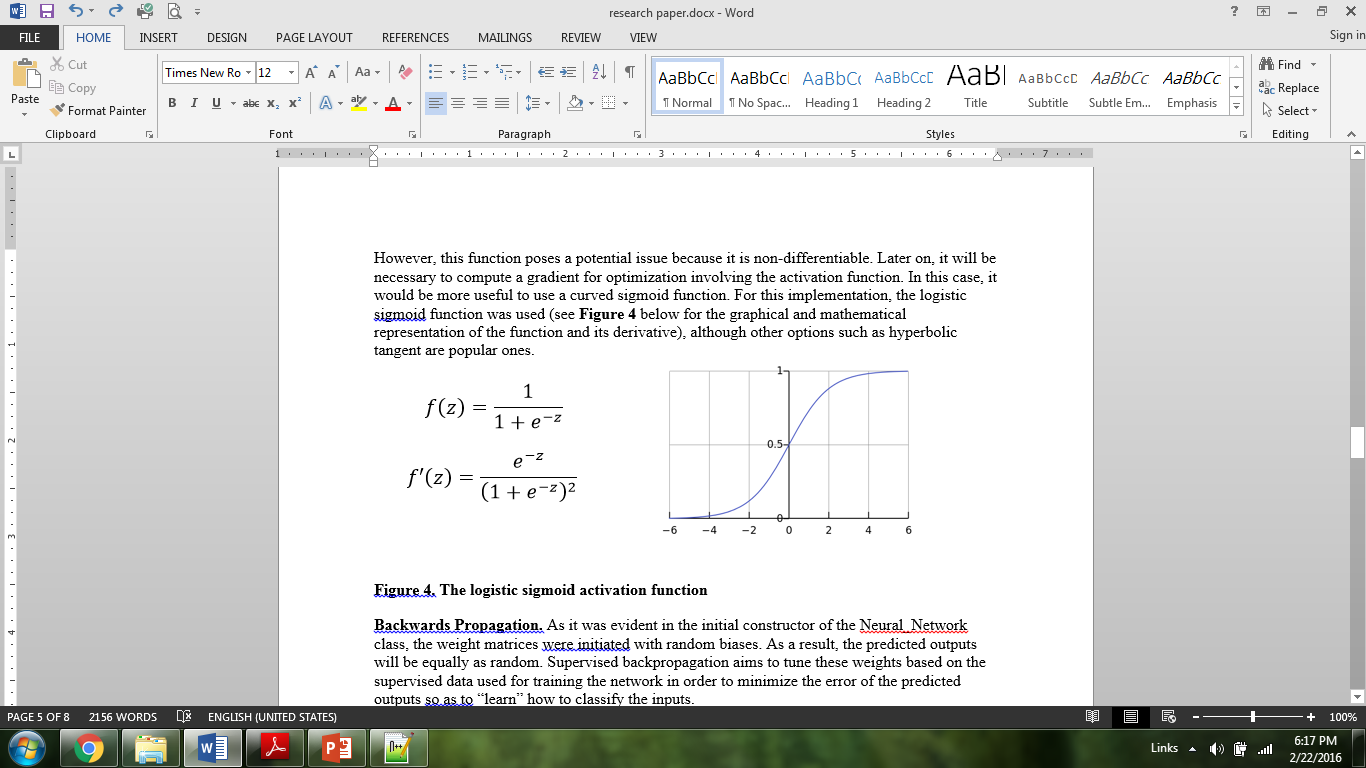
yHat **=** self**.**sigmoid**(**self**.**z3**)**

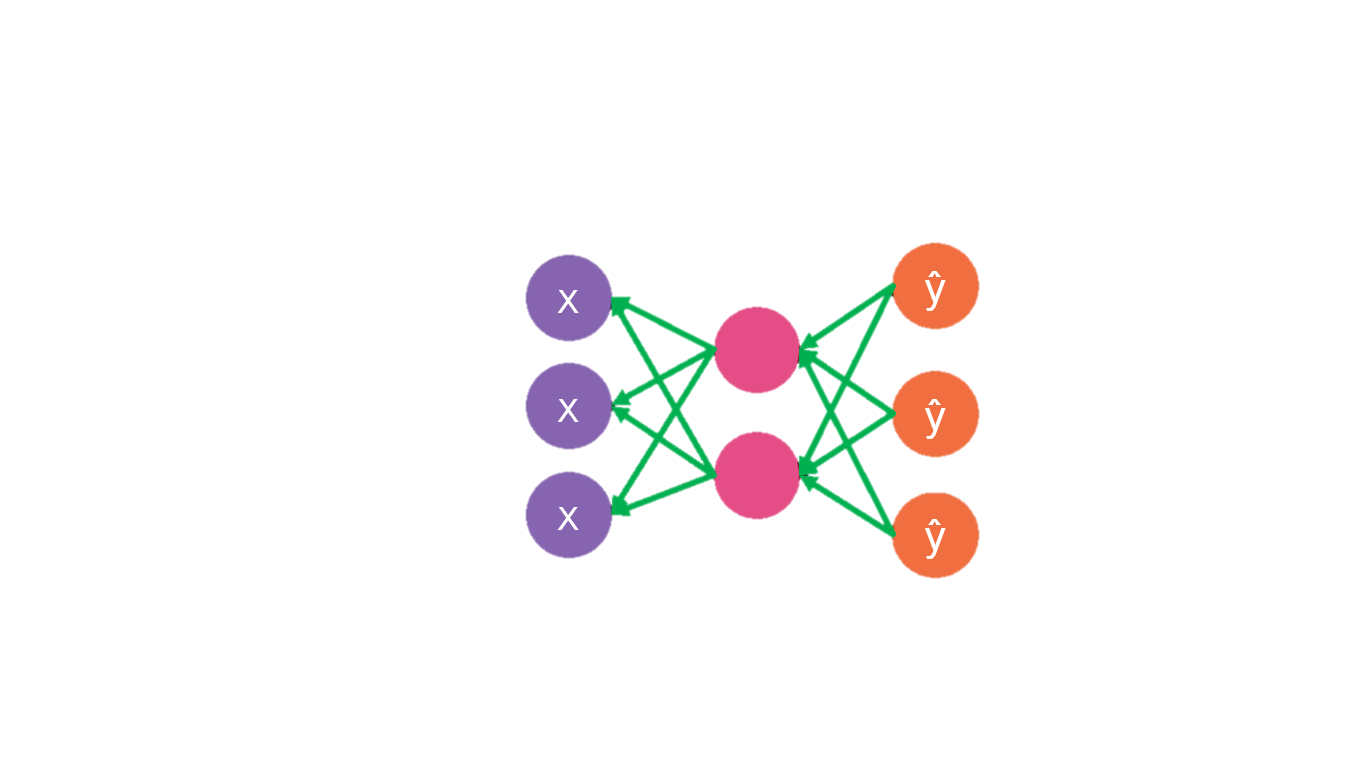
**return** yHat

**Sigmoid Activation Function.** The role of the activation function in Equations 2 & 4 is to introduce non-linearity to the forward propagation of an ANN. In the absence of an activation function, any weighted ANN could easily be condensed to a single layer neural network that is linear in nature. However, the purpose of the activation function is to allow non-linearity so multivariate learning can occur. Based on this criteria, a the fairly elementary step function shown in **Figure 3** below would be appropriate:

**Figure 3. Step function in range 0 to 1**

However, this function poses a potential issue because it is non-differentiable. Later on, it will be necessary to compute a gradient for optimization involving the activation function. In this case, it would be more useful to use a curved sigmoid function. For this implementation, the logistic sigmoid function was used (see **Figure 4** below for the graphical and mathematical representation of the function and its derivative), although other options such as hyperbolic tangent are popular ones.

**Figure 4. The logistic sigmoid activation function**

**Backwards Propagation.** As it was evident in the initial constructor of the Neural\_Network class, the weight matrices were initiated with random biases. As a result, the predicted outputs will be equally as random. Supervised backpropagation aims to tune these weights based on the supervised data used for training the network in order to minimize the error of the predicted outputs so as to “learn” how to classify the inputs. See **Figure 5** for the barebones model.

**Figure 5. Backwards propagation model**

**Cross Entropy Cost Function.** In order to tune the biases accordingly, the error of the predicted outputs must first be quantified. One approach to this problem is to calculate the variance of each predicted output like so (where N is the total number of training data):

|  |  |
| --- | --- |
|  | [6] |

However, although this approach works considerably well for continuous predictions, it falls short in supervised classification for a number of reasons. This is so because in classification problems, the outputs more likely represent Booleans of 0 and 1 or -1 and 1 using 1-of-N encoding (will be discussed later). In this sense, each datum can only have one correct classification, however the mean-squared cost function above puts a disproportionate emphasis on the error of the incorrect classifications whereas it is most important to note the error on the correct classifications. Thus, the better solution for supervised categorical problems such as image classification is the cross-entropy cost function:

|  |  |
| --- | --- |
|  | [7] |

The first term in this summation computes the error on the output neuron that has been supervised as the ‘correct’ neuron for classification (the neurons supervised with 1s). The second term does the same for the neurons that have been deemed ‘incorrect.’ However, in the second term, the cross entropy function takes the natural log of 1 minus the predicted output and thus minimizes the significance of the incorrect output neurons as this is a single-classification problem. The code for the cross entropy function is like so (ignore the regularization constant for now):

**def** cost**(**self**,** x**,** y**,** outPut**=False,** test**=False):**

self**.**yHat **=** self**.**forward**(**x**)**

**if** outPut**:**

**print** self**.**yHat

J **=** **(-**1.0**/**len**(**x**))** **\*** sum**(**sum**(**y **\*** np**.**log**(**self**.**yHat**)**

**+** **(**1**-**y**)\***np**.**log**(**1**-**self**.**yHat**)))**

#Regularizes cost function to prevent overfitting

regularize **=** **(**self**.**Lambda**/**2.0**/**len**(**x**))** **\*** **(**sum**(**sum**(**self**.**W1**\*\***2**))**

**+** sum**(**sum**(**self**.**W2**\*\***2**)))**

**if** test**:**

regularize **=** 0

**return** J **+** regularize

**Computing the Gradient.** In order to minimize the multivariate cost function, the following gradient must be considered:

|  |  |
| --- | --- |
|  | [8] |

This gradient can be extended to include all weight matrices up to and including Wn, where n is the number of hidden layers in the neural network plus one. In order to compute this gradient, the partial derivative of the cost function in respect to the weights must be computed. It is more straight forward to begin with the derivative in respect to W2, as per the *backwards* propagation model.

Begin with the cross entropy cost previously defined in Equation 7 and take the derivative in respect to the W2 matrix. (The range of the summation is temporarily ignored for readability).

|  |  |
| --- | --- |
|  | [7] |
|  | [9] |

Factor out the partial derivative of y-hat in respect to W2 and apply the chain rule of calculus and the equation relationships defined in Equations 1-4 to arrive at:

|  |  |
| --- | --- |
|  | [10] |

The next step is to is to evaluate the internal partial derivatives in Equation 10 using Equations 1-4 and simplify the expression within the brackets.

|  |  |
| --- | --- |
|  |  |
|  | [11] |

From here, using the definition of given in Equation 4 and the following identity:

|  |  |
| --- | --- |
|  | [12] |

It is evident that Equation 11 will simplify to the following expression like so

|  |  |
| --- | --- |
|  | [13] |

Although the subscript *n* and range of the summation was temporarily omitted for readability during this derivation, it is important to note that y and represent matrices of all their respective terms rather than singular values. Thus, the summation can be accounted for by using matrix multiplication by multiplying by a transposed version of the matrix *a* by the difference between matrices y-hat and y.

|  |  |
| --- | --- |
|  | [14] |

Thus, Equation 14 yields the final expression for the partial derivative of cost in respect to the second matrices of weights.

Computing a similar derivative for weights closer to the input layer of the neural network (in this case W1 being the only one to fit this criteria) follows an almost algorithmically similar derivation. The steps are the same except for the application of the chain rule in Equation 10, which requires further regression through Equations 1-4 in order to break it down appropriately, resulting in this equation:

|  |  |
| --- | --- |
|  | [15] |

Finally, using identical simplification methods as per the prior derivation, the summation must once again be accounted for through transposed matrix multiplication.

|  |  |
| --- | --- |
|  | [16] |
|  | [17] |

Thus, Equation 17 yields the final derivation for the partial derivative of the cost function in respect to the second set of weights. In more convoluted networks (more hidden layers), it is plain to see that a similar pattern persists. Thus, a more algorithmic/recursive approach may be used in order to back propagate through multiple hidden layers in the case of a deeper neural network. Nonetheless, the gradient of the cost of the neural network is computed with the following function:

#Compute derivative of cost function

**def** costPrime**(**self**,** x**,** y**):**

self**.**yHat **=** self**.**forward**(**x**)**

backError2 **=** **(**y**-**self**.**yHat**)/(-**float**(**len**(**x**)))**

dJdW2 **=** np**.**dot**(**self**.**a2**.**transpose**(),** backError2**)**

**+** **(**self**.**Lambda**\***self**.**W2**)/(**len**(**x**))**

backError1 **=** np**.**dot**(**backError2**,** self**.**W2**.**transpose**())**

**\*** self**.**sigmoidPrime**(**self**.**z2**)**

dJdW1 **=** np**.**dot**(**x**.**transpose**(),** backError1**)**

**+** **(**self**.**Lambda**\***sum**(**sum**(**self**.**W1**)))/(**len**(**x**))**

**return** dJdW1**,** dJdW2

**Minimizing cost using Batch Gradient Descent and the Conjugate Gradient Algorithm.** In order to minimize the cost (error) of the neural networks predicted outputs, the computed gradient is used to update the weight matrices in the negative direction. This batch gradient descent method will approximately approach a minimum. According to prior investigations into batch gradient descent, the minima approached in this algorithm are generally absolute at very large variable counts. In this neural network implementation, the scipy library is used in order to minimize the cost function. The previously computed gradients pose and significantly lesser computational cost than if an iterative derivative approximation approach was taken. In particular, the Conjugate Gradient algorithm was used in order to minimize the cost function. The Conjugate Gradient method is a quasi-newton algorithm often used to minimize gradients with large sets variables.

**2.2 Training**

In order for any neural network framework to carry out the previously explored supervised backwards propagation method, supervised training data is necessary. In conjunction with training data, backpropagation of neural networks is referred to aptly as *training*, as this is the stage where the ANN ‘learns’ to classify its data set, essentially by trial and error. For this stage, the following train method was implemented into the Trainer class, and it implements the batch gradient descent using Conjugate Gradient that was aforementioned.

**def** train**(**self**,** trainX**,** trainY**,** testX**,** testY**):**

#Make an internal variable for the callback function:

self**.**X **=** trainX

self**.**y **=** trainY

self**.**testX **=** testX

self**.**testY **=** testY

#Make empty list to store training costs:

self**.**J **=** **[]**

self**.**testJ **=** **[]**

params0 **=** self**.**N**.**getParams**()**

options **=** **{**'maxiter'**:** 100**,** 'disp' **:** **True}**

#Minimize cost function using computed gradient method

\_res **=** optimize**.**minimize**(**self**.**costFunctionWrapper**,** params0**,** jac**=True,** method**=**'CG'**,**

args**=(**trainX**,** trainY**),** options**=**options**,** callback**=**self**.**callbackF**)**

self**.**N**.**setParams**(**\_res**.**x**)**

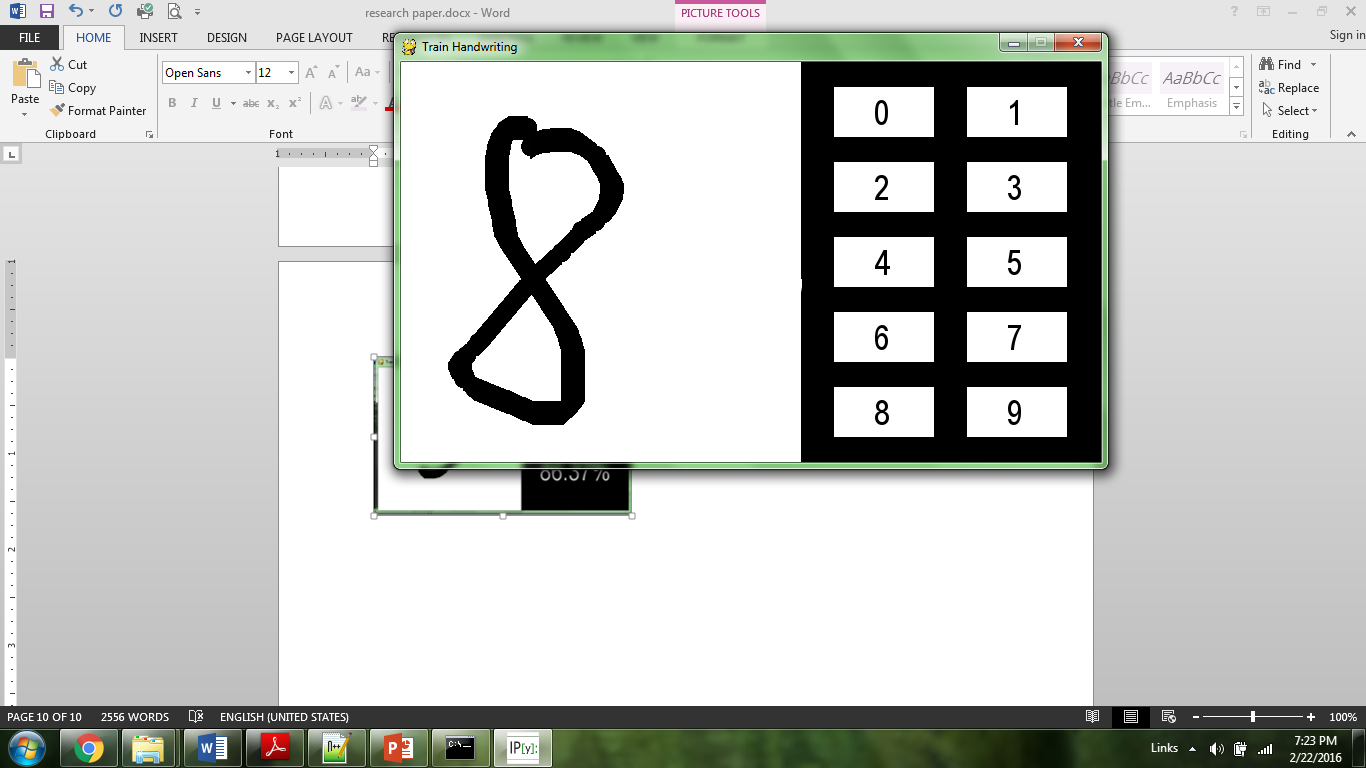
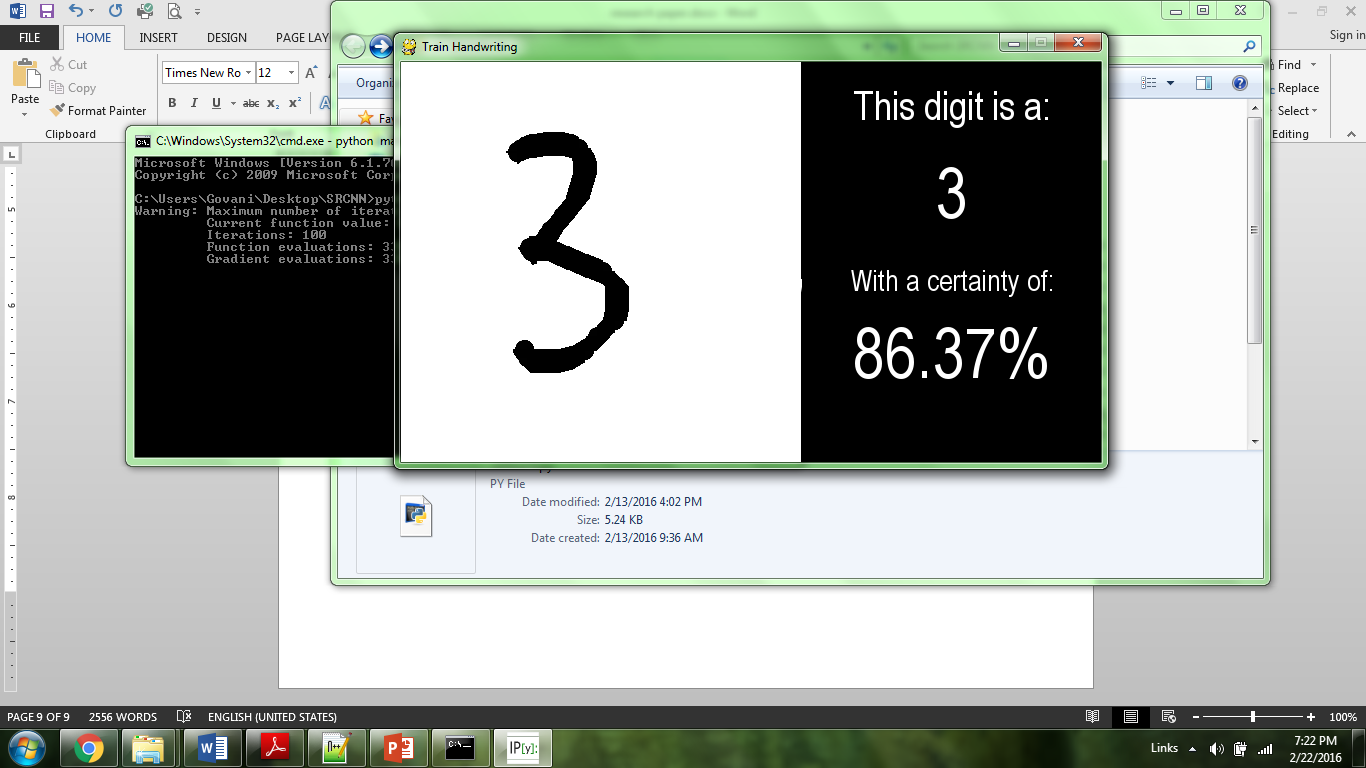
self**.**optimizationResults **=** \_res

As a proof of concept to the now built ANN, the following two training sets were implemented and tested.

**2.2.1 Optical Digit Recognition**

Optical digit recognition is an open problem due to its potential benefits in eBook digitization and computer interaction with handwritten text. In this application, the ANN is trained with supervised pixel maps of optical digit samples of the numbers from 0-9.

**Gathering pixel map data.** Rather than using available databases containing compiled pixel maps of various handwritten digits, an application was developed in order to both train the neural network and to forward propagate predicted outputs using the currently gathered pixel maps. The application was developed using pygame and can be seen in **Figure 6** below.

 **Figure 6. Pygame Application to Train (Optical) Handwriting**

The left-hand image in **Figure 6** shows an already trained network analyzing the character drawn on the white canvas using the mouse. The right-hand image in **Figure 6** shows a user training the neural network by supplying it with a drawn digit on the canvas along with a supervision through the digit selection pad on the right side of the window. When gathering the digit training data, it is important to note that it was only gathered from one source (rather than a database or multiple sources) in order to maximize experimental controls.

**Parsing the digit image data.** When collecting the pixel maps of the images, it is necessary to maintain a standard aspect ratio and pixel size in order to train a singular neural network. In order to accomplish this, a screenshot of the image on the canvas was saved temporarily. From here, the PIL (Python Image Library) was used in order to crop the image to its minimum height and width and scale it to 16 x 16 pixels. Then the image was converted into a pixel map in which each black pixel was represented by a 1 and each white pixel was represented by a 0. The pixel map was stored as a one-dimensional list. Each image’s supervised output data was stored as another tuple (derived from the file structure) and was a 10-tuple in which all indexes held a 0 except for the index of the correct output which was a 1. See the complete code for the implementation.

**Sample data file organization.** Each pixel map was stored in a separate text file as a string-tuple of length 256 (area of a 16 x 16 pixel map). Rather than have the supervision be stored within this tuple, it is preserved in the file system by its parent folder. Thus, the data folder will contain 10 folders, one for each digit 0-9.

**Integrating the Neural Network and setting hyperparameters**. Based on the format of the input data, the initialized Neural\_Network object will have an input layer of size 256 neurons (one for each pixel) and an output layer of 10 neurons (one for each digit). The hidden layer size is currently arbitrarily set, but will be optimized with data analysis later on. Finally, a parsing method was created in order to retrieve the data from the sample storage file system and convert it into matrices that are in usable formats to the Neural\_Network class. See the complete code for this implementation. The results of the ANN’s output yielded a tuple of 10

**2.2.2 Binary Diagnosis of Fine Needle Aspirates of Breast Tumors**

The second application that this neural network framework was applied to was for the diagnosis of breast cancer biopsies, particularly fine needle aspirates of breast tumors. Image data was acquired from the UCI Machine Learning Database, and it consisted of analyzed data from raw images of biopsies. Particularly, instead of raw images, this data set consisted of 30 attributes derived algorithmically from the raw biopsies. The 30 attributes were computed as the mean, standard error, and max-mean (mean of top 3) of the following ten criteria:

a) radius (mean of distances from center to points on the perimeter)

b) texture (standard deviation of gray-scale values)

c) perimeter

d) area

e) smoothness (local variation in radius lengths)

f) compactness (perimeter^2 / area - 1.0)

g) concavity (severity of concave portions of the contour)

h) concave points (number of concave portions of the contour)

i) symmetry

j) fractal dimension ("coastline approximation" - 1)

In addition to this, each data tuple was supervised with diagnosis of the tumor where ‘M’ meant malignant and ‘B’ meant benign.

**Parsing the tumor data and hyperparameters.** This image data was parsed from one data file containing 357 benign cases and 212 malignant cases as string-tuples of length 32 (one additional element was an unused ID). Thus, the Neural\_Network class was initialized with 30 neurons in the input layer and only one neuron in the output layer (with a 1 for malignant and a 0 for benign).

**2.3 Testing**