MTE 203: Project 2

Gavin Mesz

I. Introduction

The ability to recognize typed and handwritten text using computers is essential for many modern-day industries to function at their current productivity rate. It is used by the Canadian government to read official tax documents, used by computers to recognize text in ".pdf" documents, used by postal services to read name and address data and much more. The problem of recognizing handwriting from image data, although seeming simple in a human's mind, is very difficult for a computer due to the complexity of organic writing styles and natural human errors. The conventional way that computers solve this problem is often through a process called "machine learning (ML)" which allows the computer to "learn" through trial, error and correction. Although machine learning is often the answer, there are numerous ways to implement a solution for this specific problem.

This report aims to create an algorithm that trains a neural network to accurately recognize handwritten characters in the form of images.

II. Background

A. Machine Learning and Deep Learning

Machine learning is a term coined to describe computers creating their own algorithms to solve complex problems. The process is used when human mathematical derivation of algorithms seems too costly to do [1]. For example, when learning how to play chess, it is very costly for a human to develop an explicit algorithm to win. There are too many moves and strategies that one would have to memorize and account for and too many computations to do. However, a computer has the ability to easily train on thousands of chess games and quickly compute millions of mathematical calculations in real-time to determine the true optimal next moves given any game. This is exactly what IBM's Deep Blue achieved when it beat a world chess champion [2].

A subset of machine learning is called "deep learning". At high-level, deep learning is the approach to machine learning that allows computers to learn from past experiences and mistakes to develop an understanding of how to solve a problem. The computer breaks down a complex concept into a "hierarchy" of small concepts dependent on one another to give a solution. The smallest concepts are at the bottom of the hierarchy, and multiple "layers" of larger concepts are built off of these to eventually come to a full solution. This deep set of layers is where the term "deep learning" comes from [7].

For visualization, a computer might recognize handwriting by finding small edges of characters. Next, it might combine these edges into curves, curves into features and features into full characters. In deep learning, each of these steps is a layer.

B. Perceptrons

A perceptron is an artificial "neuron" inspired by the human brain and it is the basic building block of deep learning. When it was created, its primary function was to take a set of binary inputs, do a calculation, and produce a single binary output [Fig. 2].

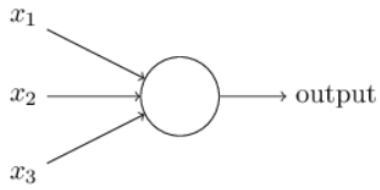

Fig. 1. Binary Perceptron [7]

Today, the perceptron has evolved to other types of artificial neurons that take in analog data. These will be from now on referred to as neurons. The neurons take a weighted sum of a set of inputs and add a "bias" value to produce an output. Their value is denoted by z (1).

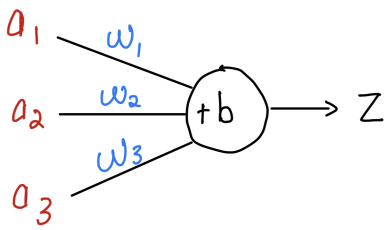

Fig. 2. Weighted Sum Perceptron

$$z = \sum_{i=0}^{n-1} w_i \cdot x_i + b \tag{1}$$

However, the brain is not made with just one neuron and the neurons have to communicate. If the language between them is not standardized, the signal becomes meaningless and mathematically, values could increase or decrease forever. Thus, activation functions are needed. The activation function normalizes the z-value of a neuron into a level from 0 to 1 signifying how much a neuron is "activated", denoted by a. There are multiple activation functions widely used but the important one that will be used in later sections is the sigmoid function (2). The sigmoid maps the input into a sigmoid curve, normalizing significantly positive or negative values of the neuron to be interpreted as an upper bound of "1" or a lower bound "0" respectively. It maps less significant values to values between the two bounds [7].

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

The sigmoid is used over other activation functions like the ReLU function when the output needs to be constrained between 0 and 1. The ReLU function takes the z-value and outputs its value if it is over 0, otherwise, it is 0.

C. Neural Networks and Multi-layer Perceptrons

When neurons are put together in layers and allowed to communicate with all neurons in the next or previous layer, a "neural network" is created. In typical "feedforward" networks or "multilayer perceptrons" where the input only propagates forward, the network is set up as an input layer, a number of hidden layers and an output layer. The input layer receives a number of inputs, the activations feed into a set of hidden layers, and those activations eventually feed through to the output layer [Fig. 3]. The highest activation in the output layer is chosen as the output of the system. The number of neurons in the input and output layer depends on the plus the output layer [6].

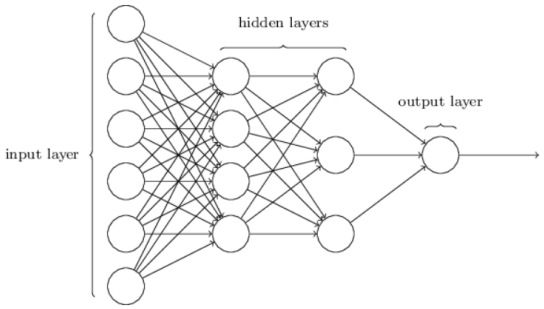

Fig. 3. Multilayer Percepton [7]

D. Feedforward Networks: Mathematical Notation and Forwards Propogation

Feedforward neural networks (FFNNs) are fundamentally driven by linear algebra and multivariate calculus. There is a vector of activation values for each layer, a matrix of weights for each set of connections between the lth layer and its previous layer, ω^l, and a vector of biases for each neuron in the lth layer, b^l. In further discussion, a^l_j will be used to describe the activation of the jth neuron in the lth layer and ω^l_{jk} will be used to describe the weight of the connection from the jth neuron in the lth layer to the kth neuron in the previous layer. Similar to the activation, b^l_j will be used to indicate the bias in the jth neuron of the lth layer. The resulting equation of the activation of a single neuron is the sigmoid activation of the weighted sum of each previous layer neuron plus the bias (3). It is important to note that the notation and understanding of neural networks have been heavily inspired by [7].

$$a^l_j = \sigma(z^l_j) = \sigma(\sum_k w^l_{jk} \cdot a^{l-1}_k + b^l_j) \tag{3}$$

In linear algebra format, the activation of a layer of neurons is a vector of (3) for all neurons in the layer. For a neural network with the structure of [Fig.4], the activation vector for layer 2 has an equation that looks like (4).

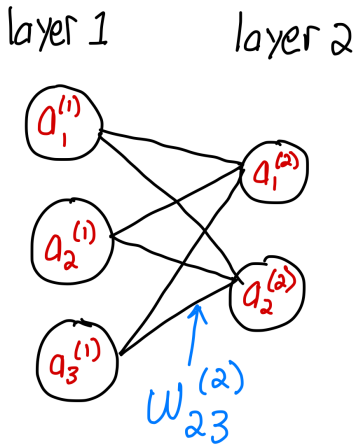

Fig. 4. Example Neural Network

$$\begin{bmatrix} \omega^2_{11} & \omega^2_{21} \\ \omega^2_{12} & \omega^2_{22} \\ \omega^2_{13} & \omega^2_{23} \end{bmatrix}^T \begin{bmatrix} a^{(1)}_1 \\ a^{(1)}_2 \\ a^{(1)}_3 \end{bmatrix} + \begin{bmatrix} b^{(2)}_1 \\ b^{(2)}_2 \end{bmatrix} = \begin{bmatrix} a^{(2)}_1 \\ a^{(2)}_2 \end{bmatrix} \tag{4}$$

This can be generalized to the form (5) where l represents the layer.

$$a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l) \tag{5}$$

Essentially, forward propagation is the repeated use of (5) until the output layer, denoted by layer L is reached.

E. Cost Function

Now that it is understood how FFNNs produce an output, the next logical step would be to understand how the network learns to produce the correct output.

The problem that needs to be solved is that the neural network has to be able to give a correct output using any given input. In terms of character categorizing, the network has to identify the correct character given an image. Essentially we have a desired output for the system, y, and an output that the system gives us, a^L.

The primary metric that will track the error or gap between y and a^L is called the cost function. The function tracks the average error between the desired and true output over multiple inputs. The goal of training the system is to minimize this function, meaning the average error between the desired and system output is minimal. There are multiple forms of cost functions that can be used in different applications. The most important one that will be addressed is the categorical cross-entropy function (6). The function takes the desired vector and multiplies it by the logarithm of the output vector [5].

$$CCE = -\sum_{j=0}^{n-1} y_j \cdot \log(a^L_j) \tag{6}$$

It is important to note that instead of a sigmoid activation function, the categorical cross-entropy function uses a "softmax" activation on the output (7). This essentially normalizes the output to probabilities on a scale of 0 to 1 which all sum to 1, signifying how confident the system is in each answer. Cross entropy uses this because the function is based on probability distributions.

$$Softmax(z^L) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{7}$$

The categorical cross-entropy function is used over a cost function like mean squared error because it converges quicker and is preferable when the neuron that is the highest value is chosen as the system output [5].

F. Gradient Descent

At the moment, the FFNN has been fed forward and it has a cost function signifying how far the network is away from its desired output. This is all that is needed to begin to make the network learn.

By adjusting each of the weights and the biases in the system to change the output, the cost function is changed. If the rate of change of the cost with respect to each weight and bias is found ($\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$), one could theoretically know how to change these parameters to optimally decrease the cost. Since the cost function is a function of all weights an biases, the instantaneous rate of change of the cost with respect to all weights and biases is just the gradient of the cost function (8).

$$\nabla C =< \frac{\partial C}{\partial w}, \frac{\partial C}{\partial b} > \tag{8}$$

It is known that the gradient gives values to change each variable to increase the function the most in one step, so if instead the variables are decremented by these values, the cost function should decrease the most in one step. Finding the gradient of the average cost and decrementing each weight and bias by this value in steps is known as "gradient descent" and it is the basis for training a FFNN. In theory, the most accurate way to go through gradient descent is to give the network all of the training data, calculate the gradient of the average cost and decrement the weights and biases by this until the cost function reaches a minimum (9).

$$< w, b > GradDescStep =< w, b > -\nabla C \tag{9}$$

However, there is a problem with using all training data for gradient descent. Although very accurate in stepping in the perfect direction, calculating the gradient using every piece of training data multiple times is very computationally intense because there could be thousands of pieces of data. Thus, an average cost is found for a "mini-batch" or small portion of training data. Calculating the gradient from this averaged error will give an approximate direction to step in to minimize the cost rather than a perfect one. This different method of gradient descent is called "stochastic gradient descent". It is not as accurate as regular descent, but it will certainly be much quicker and it will eventually find the minima. The cost function also changes instead from (6) to (10) where n is the number of training inputs.

$$C = -\frac{1}{n}\sum_n y \cdot \log(a^L) \tag{10}$$

The minimum reached by gradient descent is only the local minimum as it is impossible to find the absolute minima this way. For example, using one variable and the function $y = x^4 - 2x^3 + x$ and starting descent at $x = 1$, the process will reach only a local minimum as gradient descent cannot move up [Fig. 5].

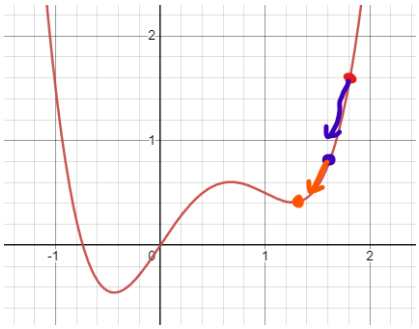

Fig. 5. Gradient Descent Limitations on Absolute Minima

The good news is that in neural network problems such as this one, it is proven that any of the local minima is a good estimate of the local minimum [3].

G. Backpropagation

Backpropagation is the act of finding the error in z-values in the output layer and propagating backwards through the neural network to derive all errors in the network. With the errors for each neuron, the mathematics to determine the gradient of the cost function is just the chain rule. There are four major steps in backpropagation. These are to first find the rate of change of the cost with respect to the z-values in the output layer, which will be called the "error" δ^L. Next, the error of each previous layer, δ^l, will be derived using the chain rule. Using those, the partial derivatives of the cost with respect to each bias $\frac{\partial C}{\partial b^l_j}$ and the partial derivative of the cost with respect to each weight $\frac{\partial C}{\partial w^l_{jk}}$ can be derived [7].

To begin, the error in the output layer is defined as the partial derivative of the cost with respect to the z-value of each output neuron z^L_j. Using chain rule and some linear algebra, the partial can be broken down into 2 parts which can be easily solved for (11). The derivation of the error in the output layer using the cross-entropy cost function came from [7].

$$\delta^L = \frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} = a^L - y \tag{11}$$

Next, the error in the output layer propagates backwards to each previous layer. The change in cost with respect to the change in the previous layer's z-value is the same as the error of the output layer times the change in output error with respect to the z-value of the previous, lth, layer (12).

$$\delta^l = \frac{\partial C}{\partial z^l} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial a^l}\frac{\partial a^l}{\partial z^l} \tag{12}$$

The partial of the cost with respect to the output layer z-values are already known as δ^L. As well, the change in the output value with respect to the activations of the previous values are all based on the transposed weight matrix (1). Lastly, the activation of any layer previous to the output layer is a sigmoid and thus the change in the activation with respect to the z-value of the neuron is just the derivative of the sigmoid (3). The change in the cost with respect to any neuron's value is simplified to (13). The circle operator represents the Hadamard product and it signifies element-wise multiplication, like a ".*" operator in MATLAB [7].

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \circ \sigma'(z^l) \tag{13}$$

Now that the effect of each neuron's z-value on the cost is known for each neuron in the network, the partials with respect to each neuron's bias and every weight can be derived. This is because the jth neuron in a layer l's value, z_j^l, is dependent on both the weight from itself to the previous neurons (indexed by k) w_{jk}^l and its own bias b_j^l (1). Therefore, the change in the cost with respect to any bias can be found using the chain rule (14).

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l(1) \tag{14}$$

Lastly, to find the change in cost with respect to the weight, a similar chain rule to finding the effect of bias is used (15).

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{15}$$

Once the rate of change of the cost with respect to the bias and weights is known, gradient descent can now be used to decrement each weight and bias by their respective partial derivative value. As explained before, the negative gradient will decrease the cost function the most in one step.

If a single mini-batch of training data is used to go through the entire forwards and backward propagation process, it is called one "epoch". If enough epochs are executed with enough training data, the cost function will reach a minimum and any new input that is presented to the network will be able to produce a desired output to a certain accuracy. For example, a network with 95 percent accuracy will produce 95 correct outputs out of 100 test inputs.

H. Training Data

It is important to note that the training data must be randomized, unbiased and plentiful. Take, for example, a network trying to classify circles, triangles and squares. The data is not random if multiple mini-batches contain only circles and some contain only squares. Training using this data will make it hard for the system to converge as it will be stepping in the wrong direction throughout training. The data is biased if the network is given only circles. If it is trained using this data then the network will only output circles. Lastly, if a network is trained on only one circle, square, and triangle then the data is not plentiful and the network will struggle to identify shapes of different dimensions. Gradient descent requires many epochs to reach a minimum and only a single mini-batch of data will not train the network enough.

III. Approach

In order to create an algorithm to train a neural network that is able to accurately recognize handwritten characters, the approach was to first start with accurately predicting handwritten numbers. This is a good step because creating a network that can distinguish between numbers 0 to 9 is a good benchmark to ensure that the algorithm used can theoretically work on a larger dataset.

Ideally, the goal is for the system to be given an image as an input and output a value from zero to nine. To do this, the structure of the neural network will be an input layer, two hidden layers and an output layer. The input layer will take in an image, the hidden layers will be for allowing the network to learn and the output layer will be a set of ten neurons each signifying outputs of zero to nine. If neuron two (indexed from 0) has the highest value, then the system has an output of two. To set up the network for forward propagation, it will be initially given a random set of weights and biases in the form of three weight matrices and three bias vectors for each layer past the input. To allow for training, a large set of training data with desired outputs will be retrieved. It needs lots of pictures of handwritten numbers with labels in order to easily give the system the desired output for each input.

The first step in using the data is to give one of the images to the input layer and propagate forward to achieve an output with random weights and biases. Throughout the network, sigmoid functions will be used as activation functions for the hidden layers and a softmax function will be used for the output layer. Sigmoid and softmax activations are used because they are ideal when each activation must be between 0 and 1 and when the output is based on the highest value respectively. Next, the cost function will be calculated using categorical cross entropy (another reason why I use softmax) and backpropagation can begin.

The four functions related to backpropagation must be implemented to allow for training. The rate of change of the cost function with respect to the output layer's z-values, δ^L, will be derived (11). Using these "errors", the rate of change of the cost function with respect to the z-values of previous layers, δ^l, will be derived (13). Using this, the derivations for $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b}$ can be derived (14), (15). These values which are effectively the gradient of the cost, will be used to minimize the cost using stochastic gradient descent. After multiple iterations of this, the system will find a minimum in the cost function with optimal weights and biases.

To test if the network is successful in training, the cost function over every mini-batch will be plotted over time as well as the system's average accuracy over time. Average system accuracy over time includes having the system make a prediction for every forward propagation and determining the average ratio of correct guesses over a mini-batch (16). The

profile of each plot will determine if the system is converging exponentially on a cost function value and, in turn, accuracy as expected from stochastic gradient descent.

$$AvgSystemAccuracy = \frac{CorrectPredictions}{MiniBatchDataPoints} \tag{16}$$

After training, the network will be tested with testing data on overall accuracy ten times. Much like the average accuracy over time, the system will make predictions upon each forward propagation and will be compared with the desired output. The total overall accuracy is the ratio of correct guesses to total guesses over the entire set of testing data (17). This ensures that the algorithm used to train the network consistently works on a group of data never presented to it. Assuming the accuracy is normally distributed, a bell curve can be created with the ten data points.

$$TotalSystemAccuracy = \frac{CorrectPredictions}{TotalDataPoints} \tag{17}$$

IV. Implementation

To create the neural network python was used as the language contains multiple libraries to deal with mathematics, data manipulation and graph plotting. To train the dataset to recognize handwritten numbers, there is a popular database of labelled handwritten numbers in the form of 28x28 pixel images called the MNIST dataset. It is a set of data that takes away foundational steps in data preparation like image processing, labelling and writing which is perfect for making sure the data is random, unbiased and plentiful to test the network training algorithm [4]. The database includes two comma-separated value tables, each with a dedicated first column of labels and beside each label is the 28x28 grid flattened into a 784 value array of grayscale values ranging from 0 to 255. There are 60000 images in the training dataset and 10000 images in the testing dataset [Fig. 6].

Fig. 6. MNIST Numbers

In the program, I used the numpy library to implement mathematical operations like matrix multiplication and manipulation during forward propagation, cost function calculation and backpropagation. I used the pandas library to retrieve the data from the comma-separated value files and lastly, I used the matplotlib.pyplot library to create graphs during the experiments.

To begin, the data was extracted from the training dataset and placed into a ”labels” vector and ”inputArray” matrix. The labels vector held every label and was the basis for the ”targetArray” matrix which held a row of ten values signifying the desired output for every input. For example, the targetArray row for a label of 7 is [0,0,0,0,0,0,0,1,0,0]. These were all kept as global variables for functions to manipulate during forward and backward propagation.

To set up the structure of the neural network, 4 vectors were created to store the values of each layer of neurons using numpy arrays. I used numpy arrays because it allows matrix algebra to be used on them. The input layer was a size of 784 to hold each grayscale pixel from the MNIST database. I used 2 hidden layers each with a size of 12 neurons. The number of hidden layers was mainly determined to be 2 layers and a low amount of 12 neurons as that would be easiest to test and visualize the algorithm created, as well it decreases the computational intensity of the network. Lastly, the output layer was given a size of 10 neurons for each output possibility [Fig. 7]. To keep track of the biases, z-values and activation values, each layer was given a separate global vector with the same size for bias value, z-value and activation value. The form of each vector is shown in (18). There were also three weight matrices created to store the weights of each set of neuron connections between layers. Each of these vectors and matrices will make backpropagation easier to code.

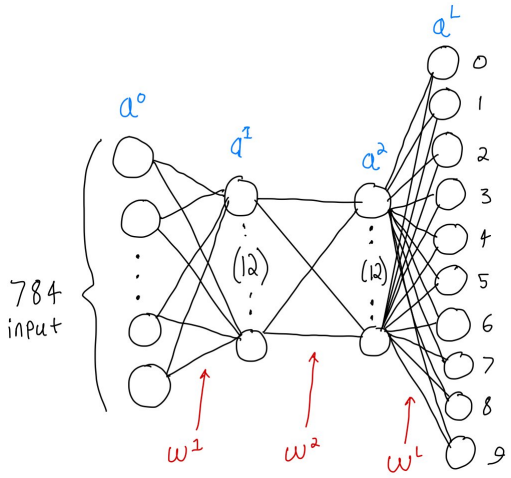

Fig. 7. Implemented Neural Network Structure Simplified

$$a^l = [a_0^l, a_1^l, \ldots, a_n^l]^T \tag{18}$$

In forward propagation, the input layer had an activation function that divided each input by 255. The MNIST data has grayscale values from 0 to 255 so doing this mapped the values linearly to a scale of 0 to 1 which will be usable by the next layer. In the hidden layers, each used (1) to determine the z-value and sigmoid activation functions (2) were used to normalize to activation values. In the output layer, the same z equation was implemented and a softmax activation was used (7). Each of the activation functions was coded to have an input of a vector and an output of the same vector but normalized.

To implement the cost function, I created another function that takes in the output layer and a desired output vector and outputs the cross-entropy cost function (10). A global list was created to hold multiple cost functions such that after a full mini-batch of training data, the mean of all stored values will be returned as an average cost of the mini-batch.

To implement backpropagation, each of the four equations from the background was implemented into the code. To solve

for the error of the output neurons δ^L, a function was created (11). It takes in the output layer and desired output layer and returns the difference between the two in the form of a vector. Next, a function to derive the rest of the layer's error, δ^l, was created based on (13). Two more functions were created to find the rate of change of the cost with respect to all weights and biases based on (14) and (15). To keep track of the partial derivatives of the cost with respect to the biases and z-values for all layers minus the input ($\frac{\partial C}{\partial b^l}$ and δ^l), a vector of the size of each layer was created to store the values. As for the partial of the cost with respect to the weights for all layers minus the input ($\frac{\partial C}{\partial \omega^l}$), a matrix identical in size to each weight matrix was created to store the values.

Once all of the functions were coded and the data was retrieved, the main program could be coded. I implemented two nested for loops, the first one signified the number of epochs and the second signifies the amount of data taken in a mini-batch. Mini-batches of 60 and 1000 epochs were chosen to use all 60000 values in the training dataset. For each epoch, the program fed an input forward, calculated the cost function, predicted a value based on the output, calculated δ^l for each layer and added what it got for $\frac{\partial C}{\partial b^l_j}$ and $\frac{\partial C}{\partial w^l_{jk}}$ to the weight and bias error vectors. At the end of an epoch, the mean weight and bias error was taken for each layer by dividing each by the mini-batch number, the average cost error was achieved in the same way. Lastly, the average bias and weight error were subtracted from the current weight and bias matrices and vectors. This process repeated for every epoch, taking in 60 images and adjusting weights and biases based on the gradient of the cost function.

I implemented the average cost over time experiment by adding a global variable called "costFns". For every epoch, the cost function value from the summation of all training data in a single minibatch is added to this variable. At the end of an epoch, the variable is divided by 60, appended to a list and reset. The list was used to plot the cost function over time once training was finished. To implement the average accuracy over time experiment, I created a "count" variable that is initialized to 0 at the start of every epoch. For each mini-batch, the network makes a prediction and if it is the same as the label, the counter increments. At the end of the epoch, the value is divided by the total number of guesses in the mini-batch and the value is appended to another list that will be used for plotting.

Lastly, to implement the total average cost of the system, I created a second array of labels, desired outputs and inputs from the MNIST testing data. Using this, I gave the network all 10000 pieces of data and made it predict the number without backpropagating. Similar to average cost over time, a count was initialized to 0 at the start of the testing and it was incremented if the network was able to correctly guess the number. At the end of testing, the value for the total number of correct guesses over the total number of guesses was displayed. This was repeated ten times and used to create a bell curve of average overall system accuracy.

The code that was used in the main program can be found in the appendix. It includes the loops I used to train the network, the loops I used to test the network and the code for the plots of cost and accuracy over time. It does not include any of the function definitions, global variables or MNIST data retrieval for better code readability and due to space requirements.

V. Results

The results of the two experiments related to the cost function over time and the average system accuracy over time were successful. Both clearly show that the cost function converges to a minimum and the average system accuracy grows accordingly [Fig. 8] , [Fig. 9].

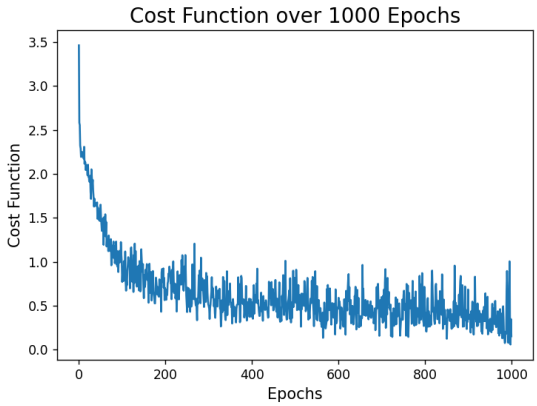

Fig. 8. Cost function over Number of Epochs

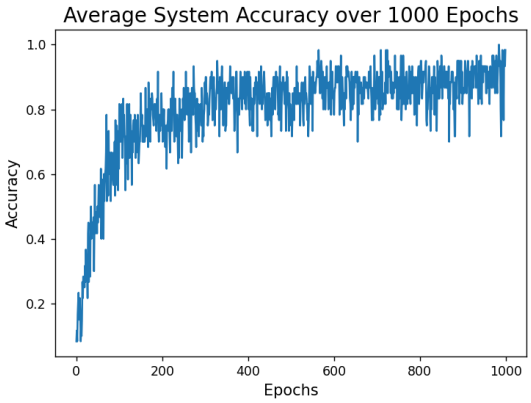

Fig. 9. Average Accuracy of the Network Over Number of Epochs

The results of the total average system accuracy were also very successful. The network using the training algorithm was able to achieve a mean of 88.9 percent accuracy. This means it was able to guess around 8890 numbers correctly out of the 10000 testing images consistently over ten tests [Table I], [Fig. 10].

TABLE I
AVERAGE TOTAL SYSTEM ACCURACY EXPERIMENTS

Experiment #	Average Total System Accuracy
1	0.8909
2	0.8904
3	0.896
4	0.893
5	0.9021
6	0.8729
7	0.8869
8	0.8909
9	0.8905
10	0.8802

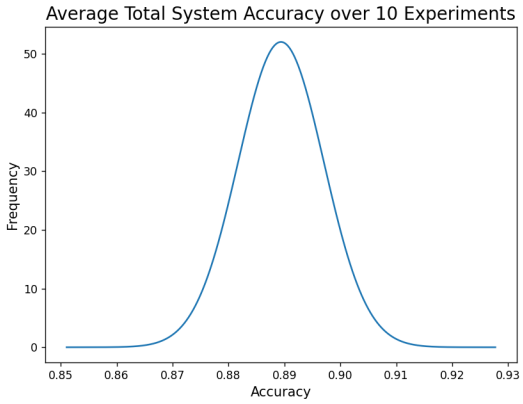

Fig. 10. Average Accuracy of the Network Over Number of Epochs

VI. DISCUSSION

The results show that the approach to training the network is successful in reliably teaching a neural network to recognize handwritten numbers. The cost function is shown to exponentially decrease over time as expected using gradient descent and the accuracy increases accordingly. Gradient descent is known to step faster the farther away the network is from the desired outputs as it is based on the cost function.

However, the problem with noise in the cost function and average accuracy of the network over time is mainly due to the same stochastic gradient descent method implemented. Each step is an approximate estimate of the total system error and each weight and bias is changed based on a relatively small sample size. Specifically, the small sample size of 60 is not a perfect estimate of the entire population of the 60000 sets of training data. This means that sometimes the gradient may be in the wrong direction, increasing the cost function instead of lowering it. Imagine the mode of a given mini-batch is the number 5. This means that the network will train to recognize the number 5 rather than step in the direction that is best for the rest of the 9 numbers, increasing the average error and, in turn, the cost. The only way to fix the noise is to increase the number of data points in a mini-batch to get a better estimate of the whole population, but this increases the computational intensity and will make the program run much slower.

An extension that was not discussed in deriving an algorithm was the concept of ”learning rate”. The learning rate is the constant multiplied to the gradient that is subtracted from each weight and bias in one step of gradient descent. Imagine the gradient shows that weight should decrease by 1 to minimize the cost function. The learning rate calculates one step as $\eta(1)$ where η is the learning rate. One could slow down the learning rate or increase the learning rate to slow or speed up convergence. However, one must also be careful to not increase the learning rate too much otherwise the solution will step too far and it could diverge. Thus, there is an optimal learning rate for the system that will allow it to converge the quickest without diverging and this is a potential for future research on optimal training algorithms.

VII. CONCLUSION

The report’s goal was to develop an algorithm to train a neural network to recognize handwritten characters. The approach was to develop a Feedforward Neural Network (FFNN), use linear algebra for forward propagation and multivariate calculus to backpropagate. The derivations from backpropagation were used to execute gradient descent to minimize the cross-entropy cost function, and in turn, the error between the desired and true network results. To implement the approach, a 4-layer FFNN using python with numpy, pandas and matplotlib using the algorithm was created. Experiments conducted included the average system accuracy and cost over time as well as the total system accuracy ten times. The results show that the algorithm was successful in training the neural network to recognize handwritten integers from 0 to 9 with an overall average accuracy of 88.9 percent. Although the network was not able to solve the problem of recognizing all characters, the successful categorization of ten numbers shows that the algorithm has the potential to categorize more characters if built upon in the future.

REFERENCES

[1] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, 2014.
[2] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1), 2002.
[3] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks, 2015.
[4] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.
[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[6] J. Heaton. *Artificial Intelligence for Humans: Deep learning and neural networks*. Artificial Intelligence for Humans Series. Heaton Research, Incorporated., 2015.
[7] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com/.

VIII. Appendix: Simplified Code

```
#TRAINING DATA
#initializing training array, contains flattened 28x28 grid of target data
initArray(labels,targetArray,60000)

#Mini batch and epoch size
minibatch = 60
epochs = 1000

#Cost Function and accuracy plots
plting=[]
plting2 =[]

for j in range(0,epochs):
    count=0
    costFns = 0
    for i in range(0,minibatch):
        index = i+j*minibatch
        feedforward(inputArray[index][np.newaxis].T)
        singleCostFn(outputlayer,targetArray[index][np.newaxis].T) #cost for single output
        if(predict(outputlayer)==labels[index]):
            count+=1
        #Calculate error for all layers
        calcError(targetArray,index)
        #add to total bias and weight error
        biaserror()
        weighterror(index)

    meanerror(minibatch) #Take mean of all weight and bias error, divide by 60
    multipleCostFn(minibatch) #Divide costFn variable by 60
    backpropogate() #decrement all weight, bias variables by mean weight/bias error
    #append cost and avg accuracy over minibatch to plot lists
    plting2.append(count/minibatch)
    plting.append(costFns)

plt.subplot(121)
plt.plot(plting)
plt.xlabel("Epochs", fontsize = 12)
plt.ylabel('Cost Function', fontsize = 12)
plt.title('Cost Function over 1000 Epochs', fontsize = 16)
plt.subplot(122)
plt.plot(plting2)
plt.xlabel("Epochs", fontsize = 12)
plt.ylabel('Accuracy', fontsize = 12)
plt.title('Average System Accuracy over 1000 Epochs', fontsize = 16)
plt.show()

#TESTING DATA
for i in range(0,10000):
    feedforward(inputArray2[i][np.newaxis].T)
    if(predict(outputlayer)==labels2[i]):
        count+=1

print(count/10000)
```