Handwritten Digit Recognition

Caleb Powell*
cmp0132@auburn.edu
Auburn University
Auburn, Alabama, USA

Savannah Sawyer†
slc0076@auburn.edu
Auburn University
Auburn, Alabama, USA

Atanu Guha‡
akg0054@auburn.edu
Auburn University
Auburn, Alabama, USA

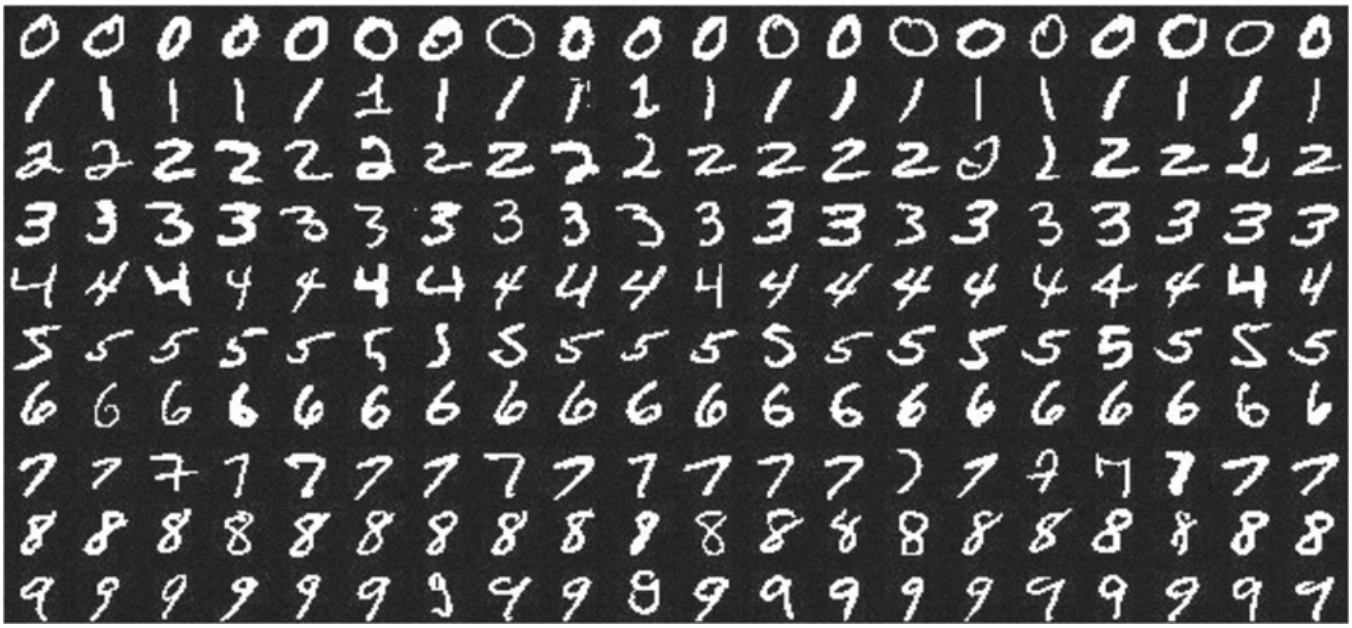

Figure 1: Sample of the MNIST data. [5]

ABSTRACT

Digit Recognition is a noteworthy and a very important topic in machine learning. As manually written digits are not always a consistent size, thickness, position and direction. In this manner, various difficulties are considered to determine the issue of handwritten digit recognition. The uniqueness and assortment in the composition styles of various individuals' handwriting additionally influence the diversity in the handwritten digits. Our project provides the implementation of the logic for perceiving and deciding these handwritten digits. The aim of this project is to implement a classification algorithm to recognize the handwritten digits. This project implementation can read in handwritten digits and computationally decipher what the digit is. This can be beneficial to teachers, people in business, blind people, mail delivery people, the entertainment world of games, and almost any person who have a hard time reading someone's handwriting. Our project uses a public data-set from MNIST database of 28x28 pixel handwritten images. Our implementation reads these images in, "runs" them through our neural network (MLP) then outputs a prediction of what the handwritten digit is. Our results were great with 93.24% accuracy using our optimal hyper-parameters and the ReLu baseline function. Although, we tried another different benchmark baseline implementation, discussed later in the report.

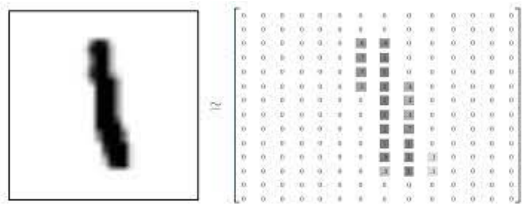

Figure 2: 28x28 pixel Image Array.

KEYWORDS

Datasets, Neural Networks, Forward Propigation, Sigmoid, ReLU, Learning Rate

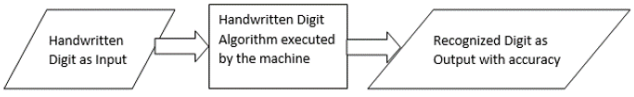

Figure 3: Functionality of the MLP Algorithm.

1 INTRODUCTION

Machine learning plays an important role in computer technology. With the uses of machine learning, human efforts can be reduced in recognizing, learning, predictions and many more areas. This report presents a project that recognizes the handwritten digits (0 to 9) from the MNIST dataset, comparing classifiers on the basis of performance, accuracy, and specificity of using different hyperparameters with the classifiers by tuning these values for optimal accuracy.

A human learns to perform a task by practicing and repeating it again and again until it memorizes how to perform the tasks. Then the neurons in his brain automatically trigger and they can quickly perform the task they have learned. Machine learning is also very similar to this. It uses different types of neural network architectures for different types of problems. Through this neural network or multilayer perceptrons, the layers are all intertwined, enabling the layers to computationally update one another. This enables the network to "learn" and recognize the handwritten digits.

2 SOLUTION DESCRIPTION

The machine will accept the handwritten digits (0-9) as inputs of 28x28 pixel images which will be passed through the algorithm we have written. After running through the algorithm, which is explained in detail in the MLP implementation and Software documentation section of this report, it produces an output of an array of confidence levels for each digit from 0 to 9 with up to 93.24% accuracy; I.e. the output will be an array of size 10 where the value at each index represents the percentage likelihood that the input image is a certain digit. For example, if the value at index 6 is a 0.97, there is a 97% chance that the handwritten digit is a 6. The process inside our algorithm will run several times until a desired level of accuracy is reached.

2.1 Users

Digit recognition system is the working of a machine to train itself or recognizing the digits from different sources like emails, bank cheque, papers, images, mail-man etc. and in different real-world scenarios for online handwriting recognition on computer tablets or system, recognize number plates of , numeric entries in forms filled up by hand and so on.

People that have problems seeing properly could use this classifier and get helped tremendously using a computer that converts handwritten images and a separate software (siri, google, etc.) that read aloud the text conversion. The entertainment world could have a use for this classifier where machines will automatically recognize the handwritten digit. This can be used to input answers in game applications. Departments of business reading handwritten quotes, expense reports, balance statements, time sheets, payroll, Invoices, etc. can also benefit from this implementation. Other users of this classifier could be teachers dealing with handwritten homework, assignments, tests, grades, etc.

2.2 Importance

This implementation is important for many reasons. One of which is that it saves time because it can automatically/computationally read thousands of handwritten digit images and convert them to readable digital text. Not only would it be exponentially faster than manually converting them, but the predictions/educated guesses of unclear handwritten digits, in comparison to the manual conversion, would be far more accurate. Mastering this implementation is important because it can be the foundation for converting all handwritten images. The study could be expanded into reading handwritten characters of any type (numbers, letters, special characters, etc.) Anything that is done by the machine without any human intervention and with more than desired accuracy is always desirable as that leads to the automatization of any process.

2.3 Dataset

We used the public database "Modified National Institute of Standards and Technology database (MNIST)." and downloaded it from the URL: http://yann.lecun.com/exdb/mnist/.[3]We have imported this dataset into python and formatted it using a keras [1] package. This loads the data in as an array. This dataset stores input the images in a 28x28 array where the value of index is between 0,255 and represents the black level of the pixel. In our implementation, we changed these values to be between 0,1 and converting the 28x28 array into a single array of size 784. As mentioned previously, the

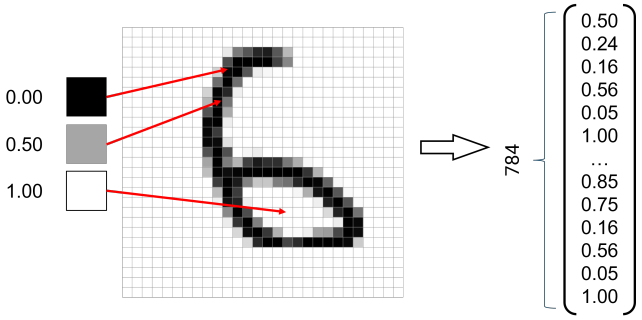

Figure 4: Example of values of a single dataset Image.

images in the dataset are converted to an array. i.e. A computer sees an image as an array of numbers. The matrix on the below right image contains numbers between 0 and 255, each of which corresponds to the pixel brightness in the left image (zero being pure black and 255 being pure white). Both are overlaid in the middle image. So, the input image becomes a 28x28 matrix.

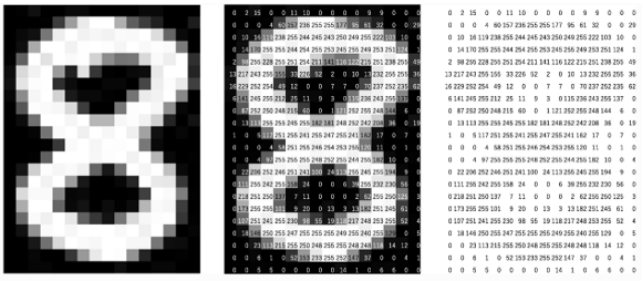

Figure 5: Structure of the training data.

The output will be a probability distribution on which of the 10 digits the image is most likely to represent. The output is an array of size 10 where the value at each index represents the percentage likelihood that the input image is a certain digit. I.e. if the value at index 6 is a 0.97, there is a 97% chance that the handwritten digit is a 6. Our steps for implementing this was to import the dataset

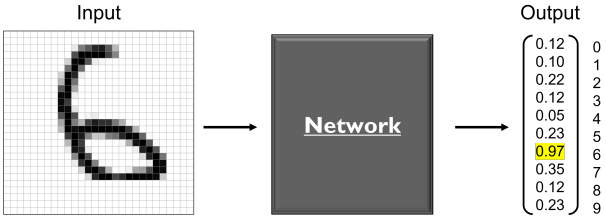

Figure 6: Functionality of the MLP.

from the keras package, reshape the X data to combine the 28x28 pixel array into a single dimension, and format the X data so the values are in the range 0,1. We trained the data on 60,000 images and we tested the data on 10,000 images.

```
from keras.datasets import mnist

# import data
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# format data
x_train = x_train.reshape(60000,784)/255
x_test = x_test.reshape(10000,784)/255
```

Figure 7: How the data was loaded and formatted.

3 MLP THEORY AND IMPLEMENTATION

MLP stands for Multilayer Perceptron. It is layers of perceptrons (input layer, at least one hidden layer, and an output layer) that make up an entire neural network. Each layer in this neural network feeds forward to the next layer and feeds backward to the previous layer, ensuring that all neurons in each layer are interconnected. For our implementation of this project of using MLP, we have 4 layers: input layer, 2 hidden layers, and an output layer. Since our inputs are 28x28 pixel images, the input layer will have 784 nodes (28x28). The amount of neurons in the hidden layers can be arbitrary; it can be any positive integer for either/both hidden layers. After a lot of tuning, we found that 350 neurons in both hidden layers had the best results. The explanation of why and how we got this number will be discussed further in the data analysis section of this report. Since the output layer is intended to be a probability distribution of the possible 10 digits ranging from 0-9, the output layer will have 10 neurons.

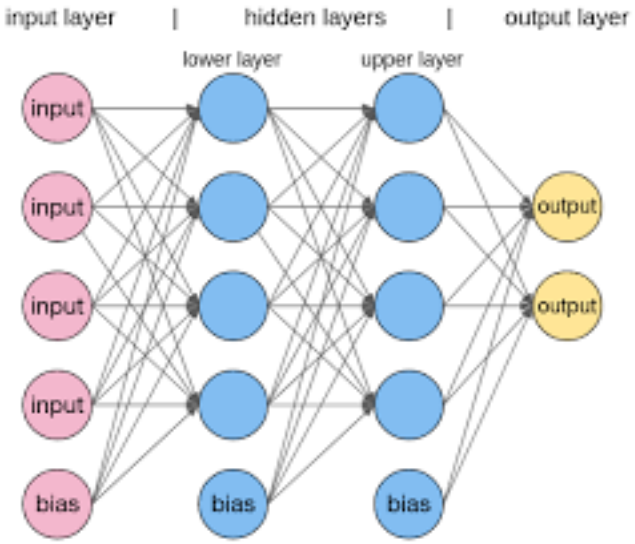

Figure 8: Depiction of an MLP.

4 SOFTWARE DOCUMENTATION

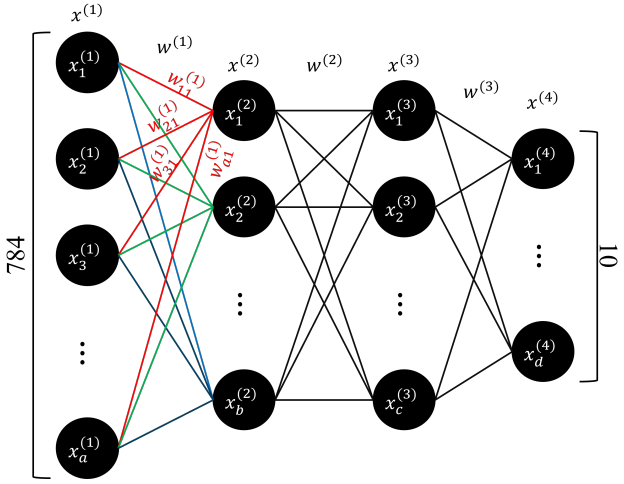

Figure 9: Neural Network Used in This Project.

4.1 Forward Propagation

In order to find the values for the nodes in the second layer $x^{(2)}$, we will use the values found in the first layer $x^{(1)}$ and perform a forward propagation. What this means is we will perform a sum on all sub values $x_i^{(1)} \in x^{(1)}$ where $x_i^{(1)}$ denotes the ith value within the $x^{(1)}$ array. Each of these values will also be weighted by the "weight" values associated with each connection in our fully connected neural network. For example, to calculate the value of $x_1^{(2)}$, we will perform the following weighted sum where $w_{ij}^{(1)}$ denotes the value of the weight that goes from the ith value in $x^{(1)}$ to the jth value in $x^{(2)}$.

$$w_{11}^{(1)} x_1^{(1)} + w_{21}^{(1)} x_2^{(1)} + \cdots + w_{a1}^{(1)} x_a^{(1)}$$

In Figure 9, you will find a visual representation of this exact weighted sum. Once the weighted sum is performed, we will pass it's resulting value into an "activation function". What exactly this activation function is will be discussed later in this report. For now we will simply state that is squeezes values passed into it within a desired range. We will refer to this activation function as ϕ.

$$x_2^{(1)} = \phi\left(w_{11}^{(1)} x_1^{(1)} + w_{21}^{(1)} x_2^{(1)} + \cdots + w_{a1}^{(1)} x_a^{(1)}\right) + b^{(1)}$$

As you can see from the above equation, we have also added a value $b^{(1)}$ to our weighted sum. This value is referred to as a "bias". Simply put, it's purpose is to make small changes to the values returned by the activation function. This bias serves a very similar purpose to that of the weights.

Now that we have shown how to calculate the values for an individual node, we can easily transition this same concept to a matrix multiplication that will return the **all** values in the second layer $x^{(2)}$ instead of just the second one $x_2^{(2)}$

$$x^{(2)} = \phi\left(\begin{bmatrix} w_{11}^{(1)} & \cdots & w_{a1}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{1a}^{(1)} & \cdots & w_{aa}^{(1)} \end{bmatrix} \begin{bmatrix} x_1^{(1)} \\ \vdots \\ x_a^{(1)} \end{bmatrix}\right) + \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_b^{(1)} \end{bmatrix} = \begin{bmatrix} x_1^{(2)} \\ \vdots \\ x_b^{(2)} \end{bmatrix}$$

Take some time to carry out this matrix multiplication and you will see how this is true. Once you have accepted its truth, we can create a shorthand notation for this matrix notation:

$$x^{(2)} = \phi\left(w^{(1)} x^{(1)}\right) + b^{(1)}$$

If we wanted to generalize this function for any layer number, we can define the following general equation:

$$z^{(n)} = w^{(n)} x^{(n)} \tag{1}$$

With that we now have the ability to calculate the values at each node given the value of the previous nodes.

$$x^{(n+1)} = \phi\left(z^{(n)}\right) + b^{(n)} \tag{2}$$

4.1.1 Stochastic Gradient Decent. In order to speed up the calculations of the gradient decent, we will perform what is known as the Stochastic Gradient Decent. What this entails is truncating the data to some arbitrary "batch size" at each epoch. After the data is truncated, it will be shuffled. After the data has been shuffled, we will perform our training on that chunk. This increases speed tremendously but decreases accuracy slightly. This will be discusses in more detail later in the report. You can see how this shuffling was implimented in the code by referencing Figure **??**

```
# Forward Propagation
z1 = x1.dot(W1) + b1
x2 = phi(z1)

z2 = x2.dot(W2) + b2
x3 = phi(z2)

z3 = x3.dot(W3) + b3
x4 = softmax(z3)

error = x4 - y
```

Figure 10: How the Forward Propagation was handled in code.

```
# shuffle the data
indices = np.arange(60000) # list from 0 -> 60,000
np.random.shuffle(indices) # shuffle the index values
X_train[indices],y_train[indices]
```

Figure 11: How the shuffling was handled in code.

4.2 Activation Functions

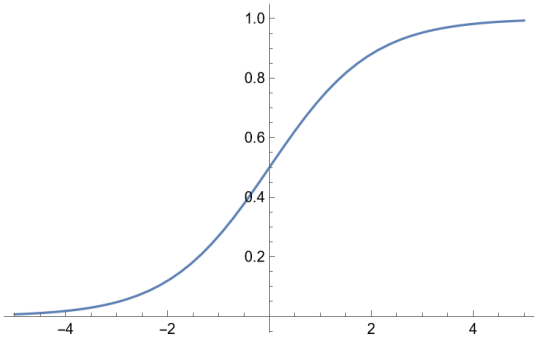

Figure 12: Plot of the Sigmoid Function.

4.2.1 Sigmoid Function.

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

The Sigmoid function is a "squish-ification" function that squeezes inputs into a range of 0,1. A plot of this is shown in Figure 24. Initially, we used the Sigmoid function as our activation function. However, we did not get very good results with this. These results will be discussed later in the report.

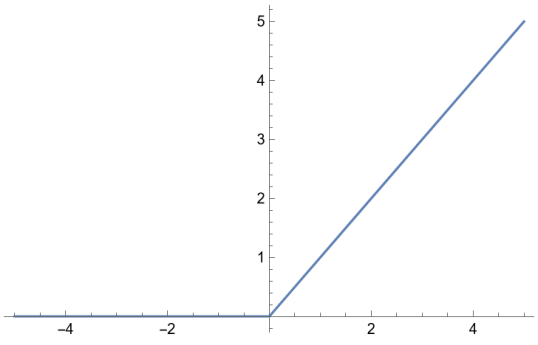

Figure 13: Plot of the ReLU Function.

4.2.2 *ReLu Function.*

$$\phi(x) = \max(0, x) \tag{4}$$

After doing some research we discovered that many Neural Networks today use the ReLU function instead of the sigmoid function [2]. A plot of the ReLU function can be seen in Figure 13.

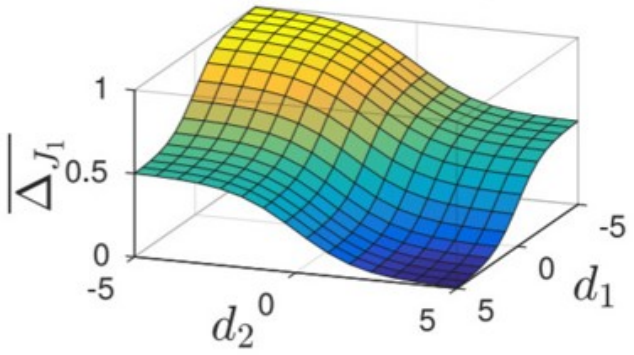

Figure 14: Plot of the SoftMax Function.

4.2.3 *Softmax Function.*

$$\phi(x) = \frac{e^{x_i}}{\sum_{n=1}^{K} e^{x_n} n} \tag{5}$$

Since we want the last layer to be a probability distribution, our activation function will be a SoftMax of all values. The SoftMax function is a normalized exponential function that converts a vector of K real numbers into a probability distribution of K possible outcomes.[4] A plot of the SoftMax function can be seen in Figure 14.

4.3 Backward Propagation

Once we have retrieved the values from the forward propagation, we will likely see that the resulting values are not what we were expecting them to be. This is because the weight and bias values are initially just random. To get accurate results, we need to adjust the weight and bias values. This may sound easy in principle but it's actually quite a challenging task that will require some multivariable calculus. Consider the following function:

$$cost = x^{(4)} - y \tag{6}$$

Here we have defined a value for a "cost". What this value represents is the difference between what our network outputs (stored in the last layer $x^{(4)}$) and the value we expected the algorithm to output (y). Consider the following example:

$$\begin{bmatrix} 0.22 \\ 0.97 \\ \vdots \\ 0.12 \end{bmatrix} - \begin{bmatrix} 0.00 \\ 1.00 \\ \vdots \\ 0.00 \end{bmatrix} = \begin{bmatrix} 0.22 \\ -0.03 \\ \vdots \\ 0.12 \end{bmatrix}$$

On the left side of this equation we can see a hypothetical output value from our algorithm. This output is a single dimension array of size 10 where the values represent the probability of the input being a particular digit. We subtract this from the expected output which should also be a size 10 array.

Now that we have defined what the "cost" is, we can make changes to the values of the weights and biases such that we minimize the value of this cost for each input. To do this we will implement a gradient decent. In layman's terms, the gradient decent is essentially taking a derivative of this cost function and making changes to the values of the weights in order to move towards the point where this derivative becomes zero. Effectively minimizing the value of our cost over time. using the gradient decent, we can defined the following rule to update the weights and biases:

$$w^{(n)} = w^{(n)} - \eta \left(\Delta w^{(n)}\right) \tag{7}$$

Here η represents the value of our learning rate and $\Delta w^{(n)}$ is the value that we get from the gradient decent. Below we have define equations for these gradients:

$$\Delta w^{(3)} = cost \cdot x^{(3)} \tag{8}$$

$$\Delta w^{(2)} = \left(E_W^{(2)} \cdot x^{(2)}\right) \tag{9}$$

$$\Delta w^{(1)} = \left(E_w^{(1)} \cdot x^{(1)}\right) \tag{10}$$

Here $E_w^{(1)}$ and $E_w^{(1)}$ represent the error in $w^{(1)}$ and $w^{(2)}$ respectively. The values for these are given below:

$$E_w^{(2)} = \left(cost \cdot w^{(3)}\right) * d\phi(z^{(2)})$$

$$E_w^{(1)} = \left(E_w^{(2)} \cdot w^{(2)}\right) * d\phi(z^{(1)})$$

Here $d\phi()$ represents the derivative of our activation functions.

```
# Backward Propagation
cost = (1/batch)*error

dz1 = dphi(z1)
dz2 = dphi(z2)

EW2 = np.dot((cost),W3.T)*dz2 # Err in W2
EW1 = np.dot(EW2 ,W2.T)*dz1 # Err in W1

DW3 = np.dot(cost.T,x3).T
DW2 = np.dot(EW2.T,x2).T
DW1 = np.dot(EW1.T,x1).T

db3 = np.sum(cost,axis = 0)
db2 = np.sum(EW2,axis = 0)
db1 = np.sum(EW1,axis = 0)

W3 = W3 - lr * DW3
W2 = W2 - lr * DW2
W1 = W1 - lr * DW1

b3 = b3 - lr * db3
b2 = b2 - lr * db2
b1 = b1 - lr * db1
```

Figure 15: How the Backward Propagation was handled in code.

4.4 Running The Code

Now that we have fully explained the mathematical theory behind the code, can actually run it. To do this, you will need to download the **MLP.py** and **functions.py** file that are linked here. These files will need to be downloaded into the same directory. The main code is in the MLP.py file. Inside this python file is a function named "*main()*". This function trains the algorithm using the following line:

$$train(epochs = 10, batch = 64, lr = 1e-3, L2N = 350, L3N = 350)$$

The output of the python file should look like this:

$$epochs : 10, lr : 0.001, L2N : 350, L3N : 350, BS : 64$$

5.73, 86.56, 88.61, 89.66, 89.97, 90.67, 91.41, 91.34, 91.77, 91.99, 92.1

What this output represents is the accuracy of our weight values after each epoch of training. In the MLP.py file, there are several commented out lines (line 26 - 52). If you un-comment these lines, it will produce **ALL** the results we got for the plotting. Keep in mind that un-commenting these lines will give you code that takes a long time to run... To get the results of Sigmoid instead of ReLU, you will need to un-comment the lines that define the activation function (line 171 and 172).

5 ANALYSIS AND BENCHMARK METHODS

In order to achieve the best accuracy, we tuned our hyper parameters: the learning rate, number of neurons in the hidden layers, number of epochs, batch size, and benchmark baseline function/implementation comparisons. For time's sake, we kept the epochs set to 10 while tuning these parameters, as each epoch takes over 30 seconds.

5.1 Learning Rate

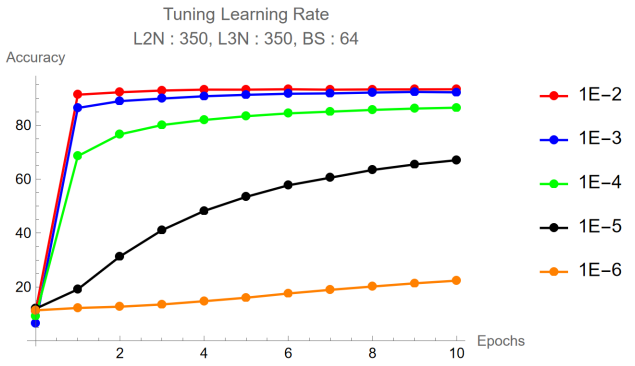

Figure 16: Learning Rate vs Accuracy Plot.

First, we tuned the learning rate. For simplicity and "easy numbers," we started with the learning rate (LR) being set to .1. At this value, the accuracy peaked at 93.36%. Next, we set LR to .01 and the accuracy peaked at 92.37%. Then, we set LR to .001 and the highest accuracy this reached was 86.52%. Next, we set LR to .0001 and the accuracy peaked at 67.04%. Lastly, we set LR = .00001 and the highest accuracy reached was 22.34%. Since the accuracy was consistently decreasing, we figured the optimal learning rate was .1 since this LR value reached the highest accuracy of 92.37% (REMEMBER: This is with only 10 epochs. Epochs will be tuned later.)

5.2 Node Count

Now that our optimal learning rate has been found, we set LR to .1 and tuned the number of neurons in the hidden layers. For relatively small, "easy" numbers, we started with the 1st hidden layer having 100 neurons and the second hidden layer having 16 neurons. These value's highest accuracy was 21.06%. Next, we changed the first hidden layer to have 50 neurons and the second to have 128. These values gave the highest accuracy of 78.94%. We then tried both hidden layers having 128 neurons, giving us the highest accuracy of 86.76%. Next, we kept the first layer having 128 neurons and increased the second hidden layer to having 256 neurons. With these values, the highest accuracy was 89.19%. We swapped the number of neurons values in the first and second hidden layer to see if that would have an impact on the accuracy, and the accuracy peaked at 89.18%; thus, this did not have much of a change. So, we continued to, instead, increase the values since that was increasing the accuracy, previously. Next, we set both hidden layers to having 256 neurons and reached an accuracy of 91.42%. We then set both

hidden layers to having 300 neurons and the accuracy peaked at 91.84%. We stuck to the pattern of keeping both hidden layers having the same amount of neurons and set both to having 350 neurons. This gave us an accuracy peaking at 92.28%. Finally, we set both hidden layers to having 400 neurons and the accuracy peaked at 92.08%. Because the accuracy was starting to decrease, we decided we had found our optimal number of neurons in the hidden layer: both hidden layers having 350 neurons. Seeing the

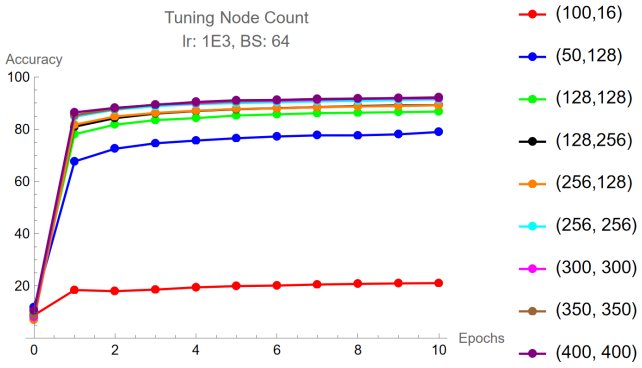

Figure 17: Node Count vs Accuracy Plot.

graph a little closer in looks like:

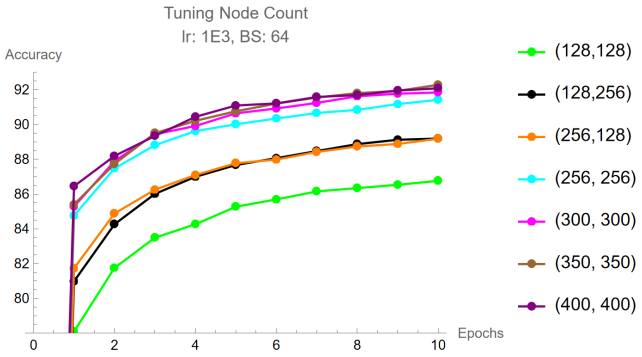

Figure 18: Learning Rate vs Accuracy Plot.

5.3 Epochs

We now have found our optimal parameter values for learning rate at .1 and number of neurons in both hidden layers at 350. Next, we found the optimal number of epochs. We increased our epochs to 50. In doing so, we created an array with accuracies at each epoch. At the end we printed this array. The following array is:

11.53, 85.22, 87.99, 89.5, 90.37, 90.91,
91.15, 91.62, 91.72, 91.85, 92.01, 92.11,
92.27, 92.43, 92.58, 92.43, 92.6, 92.46,
92.55, 92.62, 92.68, 92.48, 92.66, 92.73,
92.89, 92.75, 92.75, 92.93, 93.03, 92.8,
92.97, 93.16, 93.11, 92.91, 93.11, 93.09,
92.97, 92.94, 93.02, 93.16, 93.07, 93.05,
93.02, 93.09, 93.14, 93.09, 92.97, 93.07,
92.99, 93.02, 93.24

As you can see in the above array, the maximum accuracy reached was at epoch 50 with an accuracy of 93.24%. We figured that stopping at 50 epochs wouldn't be a bad idea, since the accuracies had been consistently high after the first epoch. Additionally, for time's sake we stopped at 50 epochs because it took over 20 minutes for the process to run 50 epochs. Also, at 50 epochs, we reached our maximum accuracy. Therefore, we chose our optimal number of epochs to be 50.

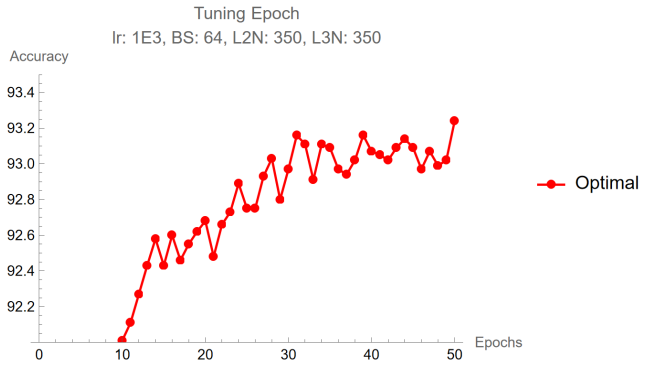

Figure 19: Epochs vs Accuracy Plot.

5.4 Batch Size

In the figures below we have analyzed how the change in batch size affects both the accuracy and runtime of the algorithm. As you can see, increasing the batch size correlates to a decrease in the accuracy. However, an increase in the batch size also correlates to a decrease in the runtime. In order to achieve a high accuracy with short runtime, we will choose a batch size around 60. We choose this value because increasing it any further will have very little impact on the runtime but substantial loss in accuracy.

5.5 Baseline Comparison

In our main classifier we used the baseline functions: ReLu, derivative of ReLu, and Softmax. We received the best accuracies with

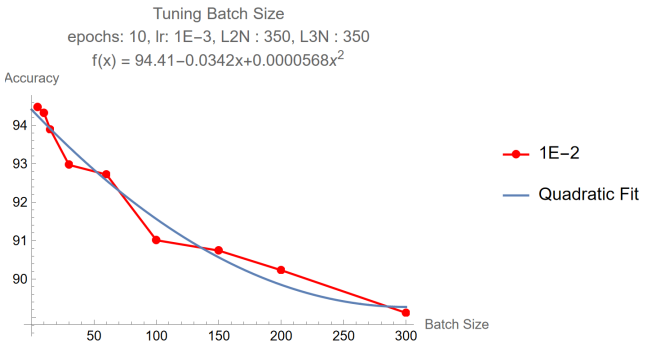

Figure 20: Batch Size vs Accuracy Plot.

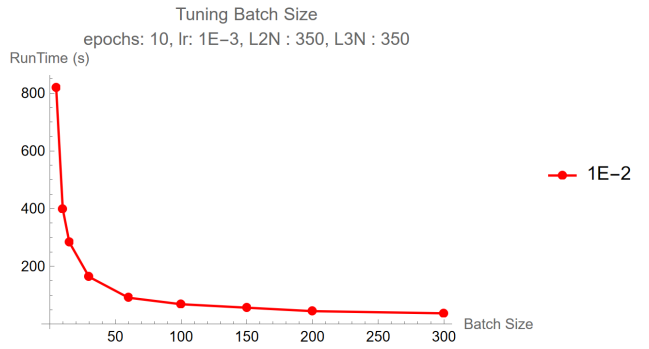

Figure 21: Batch Size vs Accuracy Plot.

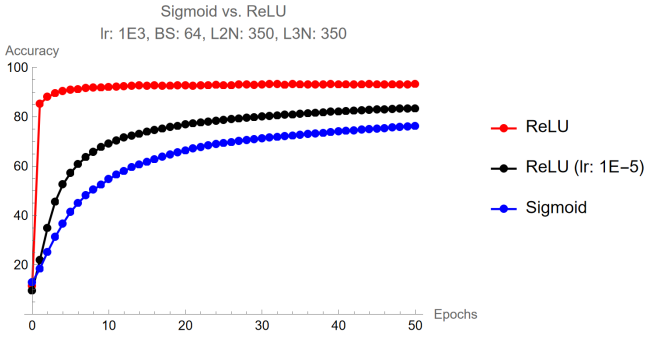

Figure 22: Relu vs Sigmoid Plot.

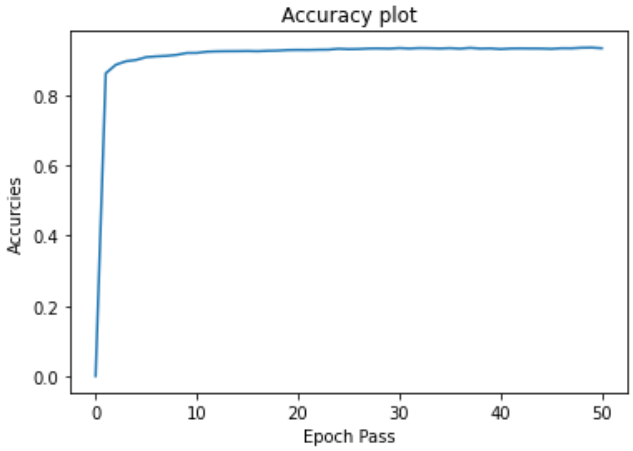

Figure 23: ReLu vs Accuracy.

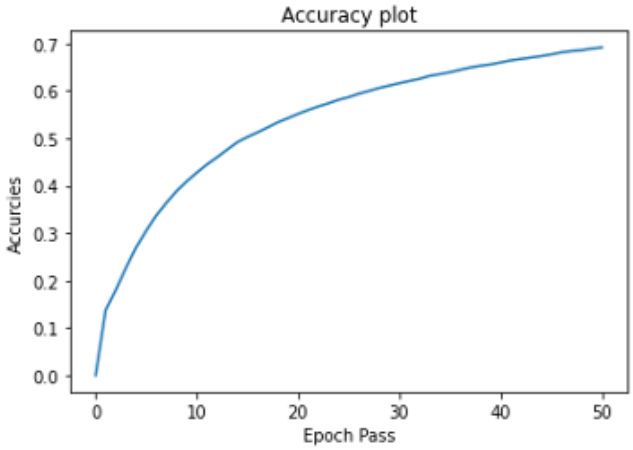

Figure 24: Sigmoid vs Accuracy.

this implementation (NOTE: these are the accuracies mentioned above peaking at 93.24%)

A benchmark we used to compare was that we implemented the baseline functions: sigmoid, derivative of sigmoid, and softmax. We tested this by using our optimal parameters of learning rate = .1, number of neurons in the hidden layers being 350, epochs being 50, and batch size being 64. With these optimal parameters being set, the accuracy peaked at 76.26%.

Thus, we believe our initial implementation of using Relu and the derivative of ReLu is much more efficient than the benchmark version of using sigmoid and the derivative of sigmoid. This is because as you can see in the graph, the accuracies were much higher with the ReLU version than with the Sigmoid benchmark version.

6 CONCLUSION

In this report, the handwritten digit recognition using machine learning methods has been implemented. We did so by importing the public dataset from MNIST. We tuned our hyperparameters of learning rate, number of neurons in the hidden layers, number of epochs, batch size, and best classifier. We found the optimal values for each of them. In doing so, we were able to return impressive accuracy results. The optimal values were LR = .1, number of neurons in the hidden layers = 350, number of epochs = 50, batch size = 64, and implementation version = using ReLu. Setting all of this hyperparamters to their optimal values gave us an accuracy of about 93.24%. We tried another benchmark version keeping all hyperpa-

Tunning LR

Epoch	LR	Accuracy
5	0.1	14.00%
5	0.01	84.40%
5	0.001	87.78%
5	0.0001	79.00%

Tunning Neurons in Hidden layers

LR	HL1	HL2	Accuracy
0%.001	100	16	56.40%
0.001	50	128	79.31%
0.001	128	128	85.67%
0.001	128	256	87.66%
0.001	256	128	87.78%
0.001	256	256	89.55%
0.001	300	300	90.30%
0.001	350	350	93.66%
0.001	400	400	91.34%

Tunning Epoch

Epoch	Accuracy	Epoch	Accuracy	Epoch	Accuracy
1	86.33%	17	92.74%	33	93.48%
2	88.77%	18	92.80%	34	93.39%
3	89.76%	19	92.96%	35	93.46%
4	90.13%	20	92.99%	36	93.35%
5	90.94%	21	92.98%	37	93.58%
6	91.16%	22	93.05%	38	93.37%
7	91.30%	23	93.07%	39	93.43%
8	91.58%	24	93.33%	40	93.23%
9	92.12%	25	93.21%	41	93.40%
10	92.14%	26	93.26%	42	93.43%
11	92.49%	27	93.40%	43	93.40%
12	92.59%	28	93.42%	44	93.39%
13	92.63%	29	93.36%	45	93.30%
14	92.64%	30	93.52%	46	93.47%
15	92.68%	31	93.39%	47	93.45%
16	92.62%	32	93.52%	48	93.46%
				49	93.66%
				50	93.47

Figure 25: Accuracy Chart.

rameters the same as before except changing the implementation version to using Sigmoid. This gave an accuracy of 76.26%. Thus, we believe our initial implementation of using the Relu version is much more efficient than the benchmark version of using sigmoid. This is because the accuracy is much higher with the ReLU version than with the Sigmoid benchmark version. Mastering the implementation of this project can be the foundation for future projects. For example, extending this same algorithm, we can apply the same technique to the other OCR applications like Letter Recognizing application.

7 CONTRIBUTIONS

- Savannah ⇒ Savannah worked on the MLP theory and Implementation section. She also tuned parameters of the actual model and plotting accuracy graphs for comparison for each hyperparameter as well as with baseline. She worked on the entire data analysis portion of this report. For each listed topic she worked on, she provided documentation for each.
- Atanu ⇒ Loaded and played around with the data. Worked on Baseline selection for accuracy comparison. Hyperparameter tuning and documentation, problem description, solution description.
- Caleb ⇒ Actual model selection, forward/backward propagation, plotted accuracies using external Mathematical tools, converted report to Latex format.

8 HARDWARE RECOMMENDATIONS

- 3.1.1 Hardware Requirements
- RAM: At least 4 GB.
- Processor: Intel(R) core (TM) i3 or more. 2.00 Ghz.
- Internet connectivity: Yes.(Broadband or wi fi)
- Webcam connectivity: Yes

REFERENCES

[1] The TensorFlow Authors. 2015. *MNIST digits classification dataset.* Retrieved November 30, 2021 from https://keras.io/api/datasets/mnist/

[2] Maciej Balawejder. 2021. *Neural Network From Scratch Using Numpy.* Retrieved November 30, 2021 from https://medium.com/analytics-vidhya/neural-network-mnist-classifier-from-scratch-using-numpy-library-94bbcfed7eae#b7ab

[3] Yann LeCun. 1998. *The MNIST Database of Handwritten Digits.* Retrieved November 30, 2021 from http://yann.lecun.com/exdb/mnist/

[4] Aditya Srinivas Menon. 2021. *MNIST Handwritten digits classification from scratch using Python Numpy.* Retrieved November 30, 2021 from https://towardsdatascience.com/mnist-handwritten-digits-classification-from-scratch-using-python-numpy-b08e401c4dab

[5] Madhu Ramiah. 2019. *How I increased the accuracy of MNIST prediction from 84% to 99.41%.* Retrieved November 30, 2021 from https://madhuramiah.medium.com/how-i-increased-the-accuracy-of-mnist-prediction-from-84-to-99-41-63ebd90cc8a0