Spring 2023

Final Project

Memphis Lau

Due: Saturday, June 10, 2023, 11:59 pm

CONTENTS

Final Project

Memphis Lau

I. INTRODUCTION

In this project, I explore using gradient descent and neural networks in machine learning. Using a provided dataset with images of hand-drawn digits, I train a model to "recognize" images of digits and classify them. With 150 epochs, 2 neural network layers, and a learning rate of 0.01, my model can accurately classify the testing set of 10,000 images 91.84% of the time. After building the model, I experiment with different numbers of epochs, layers, and learning rates to see how it affects the model.

II. THE MATH BEHIND IT

A neural network is a system that takes in an input, takes it through multiple calculations, and outputs the result. It can vary in the number of layers and dimensions of each layer. In the first layer of the neural network, the x input vector is multiplied by a set of weights and increased by a set of biases. An activation function, which depends on the context of the problem, is applied to the vector. This resulting vector then passes through the next layer, where it is multiplied by a different set of weights, increased by a different set of biases, and inputted into the activation function. In the last layer, after the initial x vector has gone through all the layers, I have a final resulting y vector or scalar. See *Fig 1* for reference.

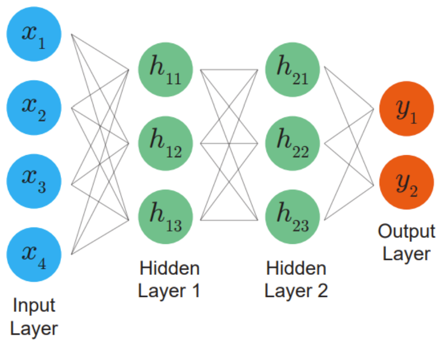

FIG. 1: Feedforward Neural Network with 2 Hidden Layers with 3 Hidden Units Each [Jaw23a]

In the context of this problem, I use a neural network to classify an image. The input vector is image data, and after going through some hidden layers, the output vector is a probability vector of the image, i.e. the probability that the given image is a 0, the probability it is a 1,...,the probability it is a 9. See *Fig 2* for reference. I can train the model using the 60,000 images of the training dataset.

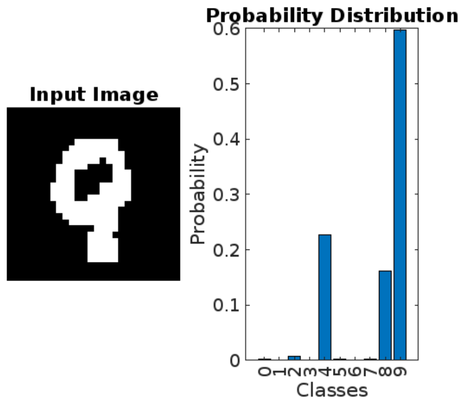

FIG. 2: Inference results on a single test image

However, how do we get the weights and biases of each layer to make this work? How do we know what is considered "right"? I am provided with the labels of all the images, meaning I have information about each image and the correct classification. Wanting the results of my model to be as close as possible to the actual labels, I can work to minimize a loss function. A loss function measures the difference between the predictions of my model and the actual answers. For the context of this problem, I will use the cross-entropy loss function (more on this in the next section). With the goal of minimizing loss, one solution could be to randomize the weights and biases an extremely large amount of times, calculate the loss function, and obtain the sets of weights and biases that minimize the loss. While this may work, this is computationally very expensive and does not provide any sense of convergence, as we saw in Homework 6.

Instead, I use gradient descent, a mathematical method used to minimize a function that usually leads to convergence with convex functions. In its simplest form, a gradient descent algorithm minimizes a function by taking steps towards the function's minimizer. These steps are calculated using the gradient of the function:

$$x_{k+1} = x_k - \alpha \nabla f(x_k) \tag{1}$$

Where α is a chosen "step size" or "learning rate". Too small of a learning rate may lead to slow convergence, while too large of one can cause fluctuations around the minimum. An important requisite of gradient descent is having an initial guess to start with.

For this specific problem, gradient descent can be used to update each set of weights and biases. As the initial guess, I randomize weights from a uniform distribution from 0 to 1 and set all the biases to be 0. In each iteration, using the gradients of the cross-entropy loss function, I update the parameters (weights and biases) of the neural network to work towards an optimal combination of weights and biases that minimizes loss, which in turn leads to a more accurate model. Using gradient descent to find these parameters is called backwards propagation.

There are generally three types of gradient descent. Batch gradient descent is when the weights and biases are updated only once after all data has been processed. This means the average gradient across all data would be used. This can lead to slow learning and potential problems not reaching the global minimum of the function. Another option is stochastic gradient descent, in which the weights are updated after every single data point is seen. However, this method is computationally expensive and vulnerable to noisy gradients. The perfect medium is mini-batch gradient descent. Splitting the training data into smaller batches, I update the neural network after each batch, ending the updates when all data points have been processed by the model. This results in computational efficiency, fast learning, and stable convergence[Opp20].

After obtaining the optimal parameters with mini-batch gradient descent, my neural network model is complete and trained. I can then apply this model onto the testing dataset, which the model has never seen before.

III. METHODS

A. Loading in Data

The first step in this project is to load the data in. To do this cleanly, I do so in a function called *load_train_and_test_data()*, which takes in no arguments and outputs four arrays:*X_train, Y_train, X_test, and Y_test.* In the original data, the images are stored as 3-dimensional objects. However, my neural network model takes in 2-dimensional arrays as inputs. Thus, to reduce each image, I compress the LxW image to a single vector of length L times W:

```
image = reshape(pixel(:,:,i), [L*W,1]);
```

Every single pixel of every single image is denoted by a value from 0 to 255, representing the intensity. To standardize this for my neural network, I also normalize each image using MATLAB's built in *normalize()* function. I reshape and normalize each of the images in the training images dataset to obtain a 784x60000 size matrix *X_train*, where each column represents the 784 pixels of an image. I repeat this process to get *X_test*, a 784x10000 matrix.

For the labels of the training set, the original data comes as a 60000x1 vector, with each entry representing the classification of the drawing. In order to calculate the loss function later, I need to first perform one-hot encoding to the labels. This is completed by creating a 10x60000 matrix, where each column represents the class of each image. Each column is all 0's except a 1 at the respective digit. For example, a 0 is encoded by $[1,0,0,0,0,0,0,0,0,0]^T$. This matrix represents the labels of all the training dataset images, *Y_train.* I repeat this process to get *Y_train*, a 10x1000 matrix.

B. Functions

Before training, evaluating, and testing my model, I first need to create all the functions needed to make my code readable and clean.

1. *initialize_parameters*

As stated in the math section, in order to perform gradient descent, I need initial weights and biases. I initialize the weights as random numbers drawn from a standard uniform distribution and the biases as 0. The dimensions of them depend on the dimensions of the layers of the neural network. Thus, my function takes in a list of layer dimensions. To go from one layer to the other, the weights matrix must be a (dimensions of second layer) x (dimensions of first layer) sized matrix. For example, if I am going from a 64-dimensional layer to a 10-dimensional layer, the weights matrix for that specific layer should be a 10x64 matrix. The biases, meanwhile, will be a single-column matrix, and the number of rows is equal to the dimensions of the following layer.

To create a matrix of random numbers, I use the *rand()* function. The biases are created using the *zeros()* function. I store all the parameters as a cell, where each struct in the cell contains the weights matrix and biases matrix of each transition between layers. To achieve this for all general cases, I use a for loop to create each struct of the cell *parameters.* The *parameters* variable will thus have L-1 cells, where L is the length of the input *layer_dims.*

2. *tanh2*

For my neural network, the activation function between hidden layers is the hyperbolic tangent function. It maps any value to the range [-1,1]. The mathematical formula for this is $\frac{2}{1+e^{-2x}} - 1$.

The input to this function is a KxN matrix X, and I want this function to apply the tanh function to every entry of X. To accomplish this, I first obtain the size of X and create a matrix Z in the shape size of X. Using nested for loops, I set each entry in Z to be the tanh function applied to the corresponding entry of X. There may be a more efficient way to accomplish this using mapping or anonymous function-defining, but this is the way I chose to do this.

3. softmax

The output of the neural network is a 10-dimensional vector, where each entry should represent the probability of the respective class. To get probabilities, I use the softmax function. This ensures the probabilities are between 0 and 1, and that the vector sums up to 1.

The formula for this is $\frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$. Similar to the process in creating the *tanh()* function, I start by initializing a matrix Z in the size of the input matrix X. Using two for loops, I set each entry of Z to be the softmax function applied to the corresponding entry of X.

4. forward_propagation

This function is essentially running an input vector X through the neural network process. The inputs are the input vector X and a *parameters* cell, where each struct of *parameters* contains the weights and biases of each layer to the next. The output is a cell called *activations*, with each element holding the layers through the neural network.

To start, I initialize *activations* as a cell of length (L+1), where L is the length of *parameters*. Of course, the first element of *activations* is set as X, the input vector. Looping from 1 to L, I can set each element of the output cell. First, I use the corresponding weights and biases to run the linear connection Weights * Matrix + Biases. Next, I run this new matrix through an activation function. If I am at the last step, the activation function is *softmax()*, to create a probability vector. Otherwise, I use *tanh()* as the activation function. The resulting matrix can be stored into the corresponding element of the cell *activations*.

5. compute_cost

To calculate the loss function, I will use the cross-entropy function. This is defined as $-\sum_{i=1}^{K} y_i log(\hat{y}_i)$, where y is the actual output and $\hat{y}$ is the predicted output from my model. Because the values in $\hat{y}$ and y are between 0 and 1, the cross-entropy loss function is better for measuring cost, as opposed to the Euclidean norm or other options.

This function inputs two matrices: AL (the model predictions) and Y (the actual output). The output is a vector of the same length as the number of columns of AL and Y. Each entry of the vector is the cost function evaluated with the corresponding columns of AL and Y. A for loop allows me to set the values: `cost(1,n) = - (sum(Y(:,n).*log(AL(:,n))));`

Where n loops from 1 to the number of columns in AL and Y. I use .* to perform element-wise multiplication with the vectors.

6. backward_propagation

This function is used to get the gradient of the loss function. The gradient consists of the partial derivative of the loss function with respect to all the variables that comprise it. The calculations of the partial derivatives can be found in *Calculation of Gradient*. The key idea is that the partial derivative of the last set of weights and biases are the easiest to find. The second-to-last set of partial derivatives depends on the last set, and so on.

This function will output a cell called *gradients*, where each struct in the cell contains fields dW and db. I start by setting the last element of *gradients* to the easily found formulas for the partial derivatives. Using a for loop that goes down from (L-1) to 1, I can find the partial derivatives of each layer and assign them to the *gradients* struct accordingly.

7. update_parameters

As stated in the math section, a value moves one step closer towards the minimizer when I subtract it from a chosen learning rate multiplied by the gradient of the function. This function completes this step.

For every struct in the cell *parameters*, I change the W field to be itself subtracted by the learning rate multiplied by the dW field of the corresponding struct of the *gradients* cell. I also change the b field to be itself subtracted by the learning rate multiplied by the db field of the corresponding struct of the *gradients* cell.

8. predict

The above functions are enough to train my model. In order to test it, I need to obtain the predicted labels of my model when given images. This function takes in the input vector X and the weights and biases of the network stored in the cell *parameters*. The weights and biases at this point will have been optimized through back propagation.

First, I run *forward_propagation()* on the input vector and the parameters, leaving me with a cell containing all the layers of the neural network. The one I am interested in here is the last layer, the y output layer. Using this layer, I find the maximum probability and use that as the model's final classification. For example, looking at *Fig 2*, the classification would be the digit 9, since it has the highest probability. I one-hot encode this to get a vector of all zeros and a single 1 in the spot of the predicted digit. Using a for loop, I can repeat this process for all columns of X to get a 10xN matrix of the predicted classifications for all images.

9. accuracy

This function checks the accuracy of my model, which will be used on the testing dataset. I first find the total number of columns, i.e. observations, in the dataset and store it. Next, I initialize a count variable as 0.

Looping through all columns, I check if the column of my model's predicted classifications match the column of the actual classifications. Because they are vectors and not scalars, I cannot use the "==" operator to check equality. I instead use MATLAB's *isequal()* function. If the vectors are the same, then I increment the count by 1. At the end of the function, I return the count divided by the total number of columns, giving me the accuracy of my model on the testing data.

10. visualize_history

This function is to be called at the end of the script, after training and testing the model. The inputs are the number of epochs used, the learning rate, the number of layers in the neural network, and two arrays representing the training loss after each epoch and the testing accuracy after each epoch.

I plot the training loss vs. epochs and the testing accuracy vs. epochs. I use the *subplot()* function in order to fit both plots into one larger figure. Using *sprintf()*, I can create a string, incorporating the provided inputs to the functions that describe the process (number of epochs, learning rate, and number of layers). This acts as my title for the entire plot. I do a similar process to create the string to save this plot into my workspace as a .png file.

C. Main Script

The main script starts with using *load_train_and_test_data()* to obtain the training and testing data for the model. The training data is a 784x60000 matrix, the labels are in a 10x60000 matrix, the testing data is a 784x10000 matrix, and its labels are in a 10x10000 matrix.

I then define the hyperparameters for training the model. This includes the input size, output size, neurons, number of layers in the network, the learning rate for gradient descent, and the number of epochs. For the first experiment, I use a neuron size of 64, 2 hidden layers, a 0.01 learning rate, and 150 epochs. Included in this section of defining parameters, I create the array *layer_dims*. This array starts with input size, has 2 elements of 64, and ends with the output size. Feeding this array into *initialize_parameters()* creates the parameters struct, with randomly chosen weights and zeros as biases.

To train the model, I use mini-batch gradient descent. I use a batch size of 64 because in a dataset of 60,000 images, it seems to be a solid size for batches. I can find the number of batches by dividing the number of images by the batch size and using *floor()* to round it off. In this step, it is also important that I initialize the trainLoss array and testAccuracy array for plotting later.

An epoch represents an entire iteration of gradient descent. The following happens in every epoch:

- I randomize the dataset's order. I do this by using *randperm()* to get indices and apply them to *X_train and y_train.*
- I initialize a cost array to store the cost after each batch.
- I loop through the number of batches. In each iteration, I run *forward_propagation()* on the corresponding batch of data and store the output into a struct called *forward*.
  - I use *compute_cost()* to calculate the cross-entropy loss between the predicted value, stored in the last element of *forward*, and the actual label in *Y_batch*.
  - I find the gradients using *backward_propagation()*.
  - I update the parameters using *update_parameters()* with the gradients.
- After all the model has seen all the batches, the parameters are optimized as much as they can be. I use *predict()* to create the matrix of predictions on the testing dataset.
- I use *accuracy()* to measure the model's performance.
- I store the norm (average) of *cost*, which has the cost function evaluated at each mini-batch, as the i-th element of the *trainLoss* array.
- I store the accuracy as the i-th element of the *testAccuracy* array.

This process repeats for as many epochs as I set in the beginning. Because the model keeps seeing more data in each epoch, the parameters continue to get closer to minimizing loss.

After all epochs have been run, I print out the accuracy of my model after all the training. This value is stored in the last element of *testAccuracy*. Finally, I call *visualize_history()* to see the plot of training loss vs epochs and test accuracy vs epochs.

IV. CALCULATIONS AND RESULTS

A. Calculation of Gradient

For this problem, I use the cross-entropy loss function, represented as $-\sum_{i=1}^{K} y_i log(\hat{y_i})$. So the gradient is calculated as:

1. Output layer:

$$\delta^{(4)} = \frac{\partial L}{\partial \mathbf{z}^{(4)}} = \mathbf{a}^{(4)} - \mathbf{y}$$

2. Hidden layer 2:

$$\frac{\partial L}{\partial \mathbf{W}^{(3)}} = \delta^{(4)} \frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{W}^{(3)}} = \delta^{(4)} \mathbf{a}^{(3)}$$
$$\frac{\partial L}{\partial \mathbf{b}^{(3)}} = \delta^{(4)} \frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{b}^{(3)}} = \delta^{(4)}$$
$$\frac{\partial L}{\partial \mathbf{a}^{(3)}} = \mathbf{W}^{(3)} \delta^{(4)}$$
$$\delta^{(3)} = \frac{\partial L}{\partial \mathbf{z}^{(3)}} = \mathbf{W}^{(3)} \delta^{(4)} (1 - \tanh^2(\mathbf{z}^{(3)}))$$

3. Hidden layer 1:

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \delta^{(3)} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{W}^{(2)}} = \delta^{(3)} \mathbf{a}^{(3)}$$
$$\frac{\partial L}{\partial \mathbf{b}^{(2)}} = \delta^{(3)} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{b}^{(2)}} = \delta^{(3)}$$
$$\frac{\partial L}{\partial \mathbf{a}^{(2)}} = \mathbf{W}^{(3)} \delta^{(4)}$$
$$\delta^{(2)} = \frac{\partial L}{\partial \mathbf{z}^{(2)}} = \mathbf{W}^{(2)} \delta^{(3)} (1 - \tanh^2(\mathbf{z}^{(2)}))$$

4. Input layer:

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \delta^{(2)} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(1)}} = \delta^{(2)} \mathbf{a}^{(1)}$$
$$\frac{\partial L}{\partial \mathbf{b}^{(1)}} = \delta^{(2)} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{b}^{(1)}} = \delta^{(2)}$$

[Jaw23b]

B. Original set of hyperparameters

First, we use 150 epochs to train the model, a learning rate of 0.01, and 2 hidden layers inside the neural network. With this model, when we train it using mini-batch gradient descent and test it on the testing data, we have an accuracy of **91.84%**. However, it is important to note that this percentage changes with every time the script is run, since the initial parameters are randomly generated, and the mini-batch gradient descent method uses randomization. This number, and all other future accuracy numbers, are simply the output on my first run of the script. *Figure 3* displays the plots of training loss vs epochs and testing accuracy vs epochs.

We see that the training loss follows an exponential decay shape, plateauing at around 70. The training loss starts off very high but drops quickly as I train the model through more epochs. The decrease in training loss becomes very small as the model goes through more epochs.

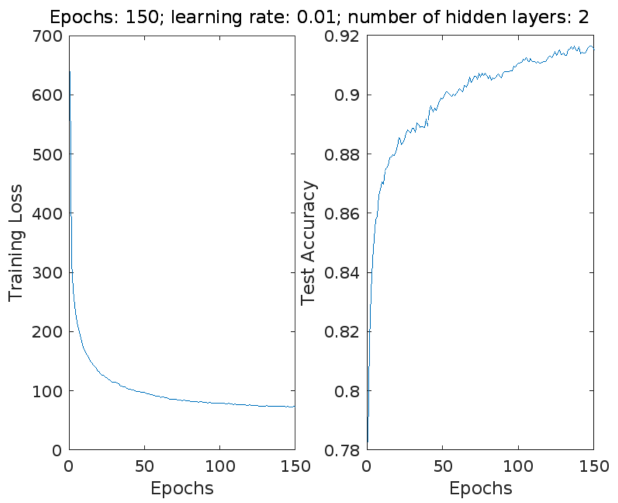

FIG. 3: Plots of Training Loss vs. Epochs and Testing Accuracy Vs. Epochs

As for testing accuracy, the shape of the plot is logarithmic, increasing significantly in the beginning and eventually slowing down. In 150 epochs, the accuracy peaks at around 92%. Both of the graphs do not follow a perfect curve; there are some oscillations. This occurs due to the nature of randomizing the data and creating batches in every epoch. A typical property of mini-batch gradient descent is the slight rigidity of the plots.

C. Changing training epochs

In order to analyze the effect of decreasing or increasing the number of training epochs, I run my script again with the same learning rate (0.01) and number of hidden layers (2), but changing the number of epochs. As seen above, when there are 150 epochs, my model has a 91.84% accuracy. *Figure 4* depicts how my model performs with 50 epochs. The accuracy after 50 epochs is **90.5%**. *Figure 5* displays the plots for my model trained with 300 epochs. The accuracy after 300 epochs is **91.17%**.

As expected, the shapes of the graphs are similar to that of the original hyperparameters. With the same learning rate and number of layers, the shape stays a similar exponential or logarithmic curve. With fewer epochs (50), the model's training loss does not go as low, and the accuracy does not reach as high. With more epochs, the model's training loss goes lower, and the accuracy surprisingly slips a little. This shows that the accuracy of my model peaks at around 91-92% and cannot increase, even as the model gets trained through more epochs.

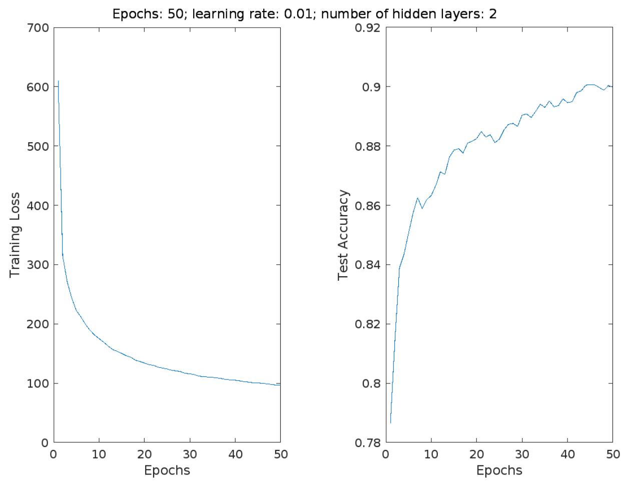

FIG. 4: Plots of the Model after 50 epochs

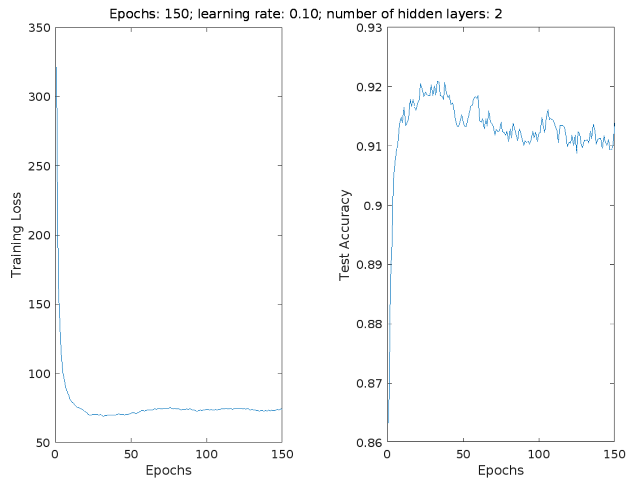

FIG. 6: Plots of the Model with learning rate 0.1

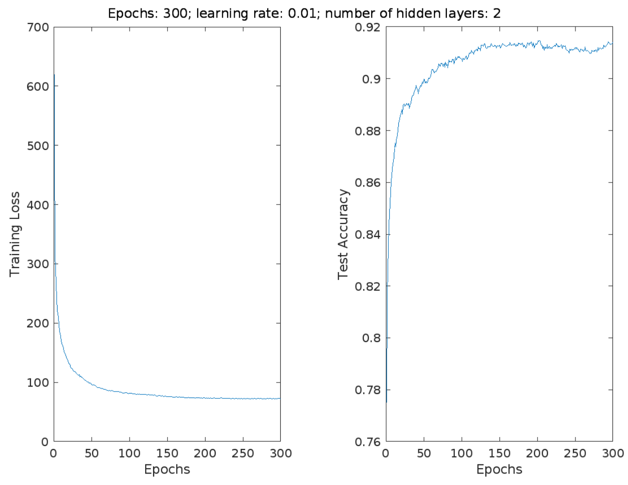

FIG. 5: Plots of the Model after 300 epochs

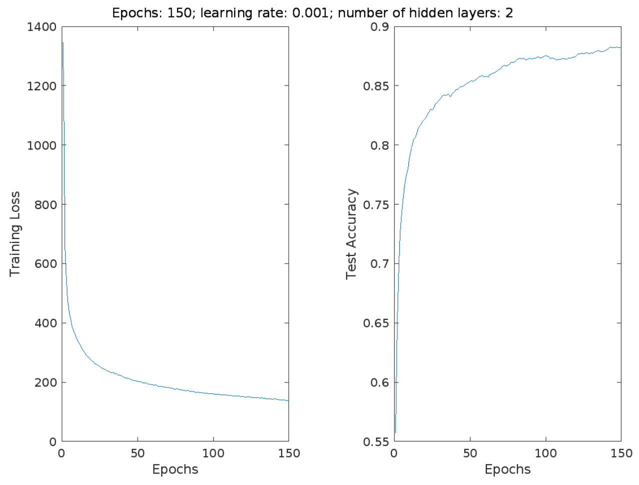

FIG. 7: Plots of the Model with learning rate 0.001

D. Changing Learning Rate

Now, I will run the same script but with different learning rates; the number of epochs will stay constant at 150, and the number of layers stays at 2. As we know, when the learning rate is 0.01, my model has an accuracy of around 91.84%. *Figure 6* shows my model with a learning rate of 0.1. The accuracy after 150 epochs of training the model is **91.39%**. *Figure 7* is my model's performance with a learning rate of 0.001. When using a learning rate of 0.001, the model has a final accuracy of **88.17%**.

When the learning rate is increased to 0.1, the training loss again decreases very rapidly in the beginning, following a logarithmic shape. However, it slightly increases as the model goes through more epochs. A similar story occurs for the testing accuracy of the model. It follows a logarithmic shape in the beginning, much like that of the model with a learning rate of 0.01. However, the fluctuations are significantly larger, and the accuracy trends slightly downward.

When the learning rate is reduced to 0.001, the training loss starts higher, at 1300+. It also decreases dramatically. However, the training loss does not drop as low as that when the training rate is 0.01. The accuracy also does not reach as high as when the learning rate is higher. The shapes of both plots are mostly similar to those of the initial hyperparameters.

E. Changing Number of Hidden Layers

Another hyperparameter to explore is the number of hidden layers in my neural network. Each hidden layer is set to have a dimension of 64 x 1. Again, I test this using a constant number of epochs at 150 and a constant learning rate of 0.01. *Figure 8* displays how my model performs with 3 hidden layers. The final accuracy after 150 epochs with 3 hidden layers is **90.86%** . *Figure 9* illustrates my model with 5 hidden layers. The accuracy is now **88.79%**.

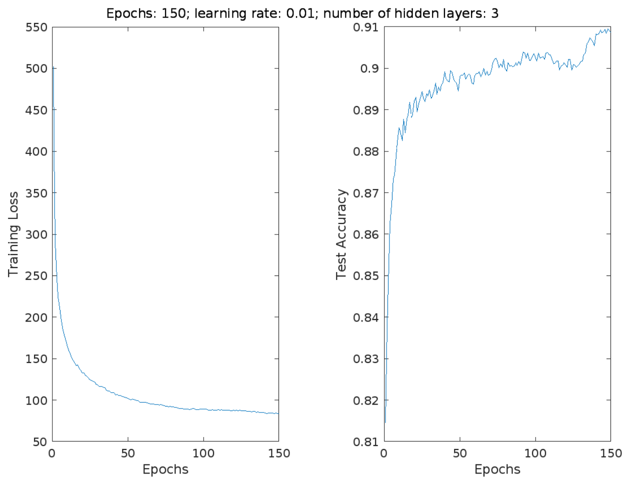

FIG. 8: Plots of the Model with 3 hidden layers

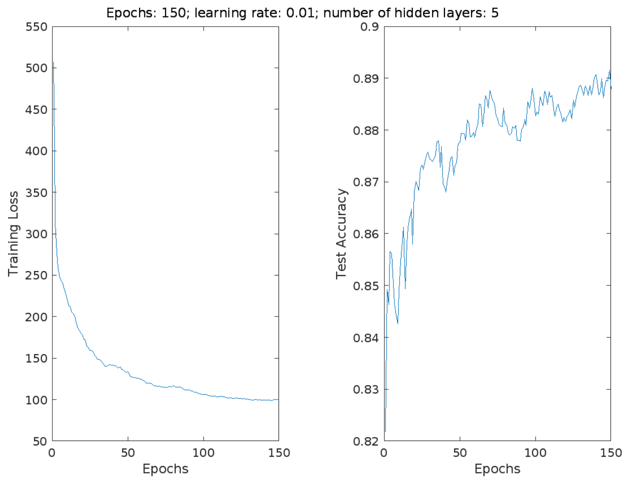

FIG. 9: Plots of the Model with 5 hidden layers

In *Figure 8*, there are 3 hidden layers in my neural networks. The shape is similar, but the plateauing happens at a higher training loss and a lower test accuracy. The training loss starts at a lower point compared to the model with the initial hyperparameters, while the testing accuracy begins higher.

The plots for my model with 5 hidden layers deviates from the normal logarithmic shape we have been seeing. The training loss vs. epochs plot almost appears quadratic or cubic; there is not as much of a dramatic fall followed by a plateau. The loss also ends at around 100 after 150 epochs, while the model with 2 hidden layers reached around 80. Again, it starts at a lower number than the original plot. The test accuracy plot also starts off higher, but does not follow a strict logarithmic shape. There are many oscillations, and the accuracy does not reach as high a value.

V. DISCUSSION

From the plots above, it is clear how changing hyperparameters can affect how my model performs.

Keeping everything else constant, changing the number of epochs does not change the shape of the plots of training loss vs. epochs and test accuracy vs. epochs. When the model goes through fewer epochs, it does not reach full convergence, and the accuracy does not reach its peak. However, somewhere between 100 and 200 epochs, my model attains its maximum performance. Even when the model is trained on 300 epochs, the training loss does not get any lower, and the test accuracy does not improve. 150 epochs seems to be an optimal number, considering time, efficiency, and performance.

The learning rate has a strong effect on my model. When the learning rate is set to 0.1, the training loss actually increases as the number of training epochs increases. The test accuracy also decreases and is very sporadic. The reason for this is that the learning rate is too large. In every iteration of backwards propagation, I update the weights and biases of the neural network using the learning rate and the gradient. When the learning rate (which represents the step size) is too large, the parameters may not be moving towards the true minimum of the loss function, but instead may jump around the minimum. This explains the variance seen in the plots. When the learning rate is set to 0.001, the training loss does not decrease as much as when the learning rate is at 0.01. In addition, the test accuracy does not peak as high. When the learning rate is too small, the convergence rate is slow. Even after 150 epochs, the model does not reach peak performance because of the small learning rate. The parameters do not step towards the minimum of the loss function quick enough.

The number of hidden layers in the neural network has an interesting impact on my model. With everything else constant, increasing the number of layers does not seem to improve my model. With 3 hidden layers instead of 2, the model still reaches a similar training loss minimum and test accuracy maximum, but it takes more epochs to reach it. The shape of the graph transitions from a logarithmic curve to almost a quadratic curve. With 5 hidden layers, the model actually performs worse, and its plots appear quadratic. This may be because with more hidden layers come more weights and biases to optimize. Perhaps this causes the model to be less accurate because of all the layers that need to be developed. However, the more likely problem is overfitting. When there are more layers, the model becomes more complex. Because the model is trained with only the training images, the weights and biases are optimized only for these training images. More layers means the model can more accurately predict the training data. However, the model may be overfitting the training data, meaning it does not perform as well on other data that it has not seen. Not all data is exactly like the training data, so when the model is so refined and dependent on the training data, its performance on testing data will not be as strong.

Overall, after discussing the changes that different hyperparameters can cause, it appears that my initial set of hyperparameters (150 epochs, learning rate of 0.01, 2 hidden layers) does a strong job of optimizing the model.

VI. CONCLUSION

To summarize, in this project, I created a machine learning algorithm in which a neural network's weights and biases are optimized through mini-batch gradient descent and backwards propagation. At the end of training my model, it can input image data of a hand-drawn digit and output a probability vector classifying the digit. When tested on a test dataset, the model accurately predicts the correct digit around 91% of the time. I tested different the model with different numbers of training epochs, learning rates, and numbers of hidden layers and concluded that the initial choice of 150 epochs, a learning rate of 0.01, and 2 hidden layers is most optimal for my model.

REFERENCES

[Jaw23a] M. Khalid Jawed. "HOMEWORK 6". In: *CEE/MAE 20 -Introduction to Computer Programming with MATLAB* (2023).

[Jaw23b] M. Khalid Jawed. "M20_Sp2023_LaTeX_Share". In: *CEE/MAE 20 -Introduction to Computer Programming with MATLAB* (2023). URL: https://www.overleaf.com/read/ygwzbytrstgj.

[Opp20] Artem Oppermann. "Stochastic-, Batch-, and Mini-Batch Gradient Descent". In: *Medium* (2020). URL: https://towardsdatascience.com/stochastic-batch-and-mini-batch-gradient-descent-demystified-8b28978f7f5#:~:text=Advantages%20of%20Mini%2DBatch%20Gradient%20Descent&text=Faster%20Learning%3A%20As%20we%20perform,a%20much%20faster%20learning%20process..