Homework 3

Logan Hall

I. Project Overview

The overall objective of this project is to use a feedforward network to solve the classification problem. Classification is a supervised learning task in machine learning that aims to predict a categorical or discrete label for a given input [3]. In the case of this project, our inout is the MNIST dataset. This dataset consists of images of handwritten digits from 0-9. This dataset is commonly used for training and testing in the field of machine learning. Within MATLAB, our goal is to classify the handwritted digits. In a broad overview, we will do this by first training the feedforward neural network on a set of training images and labels. Then we will evaluate the performance of the newly trained feedforward neural network using a seperate set of test images and labels.

II. Programming Implementation

A. Loading and Pre-Processing Data

To begin this problem, we need to load the images and labels and preprocess the data. We accomplish this in the function: `load_train_and_test_data`. Within this function, we load both the train and test data or the images and labels. This data is provided in the .mat format. Next we flatten the images by taking the 2 dimensional 28x28 array for each image and convert it into a one dimensional vector in order to make it compatible with the feedforward neural network architecture. We also normalize the pixel intensity by dividing the grayscale image values by 255, giving pixel intensities that range from 0 to 1. For the labels, we convert the values to be represented by one-hot encoding where a single unique binary digit corresponds to the label. For example, the digit 0 is represented by [1,0,0,0,0,0,0,0,0,0], and the digit 9 is represented by [0,0,0,0,0,0,0,0,0,1]. Once this has been done, the train and test data of labels and images has be loaded into our program in the desired format.

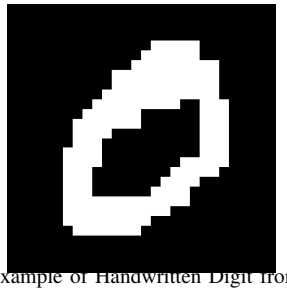

Fig. 1. Example of Handwritten Digit from MNIST

B. Setting up the Architecture and Parameters

To begin training the feedforward neural network, we need to start defining and initializing our parameters. In defining the network architecture, we set the input size to the length of the flattened image and the output size equal to the number of possible different digits, 10. We then define that there will be two hidden layers, each with 64 neurons. The hyperparameters needed to be defined are batch size, epoch, and learning rate. The batch size is defined by the number of training examples used to compute the gradients, and for us we set the batch size to be 64. An epoch is one complete pass through the entire training dataset, and we set it to 150. The learning rate determines the step size at which the weights and biases are updated during the training process, and in this project it is set to 0.01. These are the default values that we will adjust later on.

We then implement the function `initialize_parameter`. In this function, we initialize the weights and biases of the feedforward neural network. The weights are initialized to be random from a normal distribution using the MATLAB function `randn`. The biases are then initialized to be an array of zeros.

C. Training the Feedforward Neural Network

The next step in training the feedforward neural network is to do a forward pass. We start a for loop to loop through all epochs and batches within each epoch. We then carry out the forward pass in the function `forward_propagation`. Within this function, we also have defined the activation function `tanh2` and the softmax function `softmax`. The activation function introduces non-linearity into the network allowing the network to model complex relationships. The `softmax` function in turn is a mathematical function used to map the inputs to a probability distribution over the possible classes. the softmax function is given by the equation:

$$A = e^{x_i}/(\Sigma_{J=1}^{K} e^{x_j})$$

The `forward_propagation` function then runs the activation function after each layer except for the last hidden layer. For the last hidden layer, we use the softmax function. The result is the output of the `forward_propagation` function. This output is then compared to the actual output and an error is computed. This error is computed as cross entropy loss within the function `compute_cost`. We next compute the gradients of the error using a backward pass. The backwards pass follows the backwards pass algorithm, as described in the problem outline [3]. This algorithm returns the gradients of the cost with respect to each parameter in the form of a struct. Next, we update the parameters of our network using gradient descent. We accomplish this in the function `update_parameters` by taking the old weights, old biases, the gradients and learning rate to update the

weights and biases. Completing this sequence for all epochs and batches within each epoch, we have finished training the model.

D. Evaluating the Model

Now that we have trained the model, it is time to test the model. First, we define a function `predict.m`, where we return the predicted classes of the inference images. We do this by doing a forward pass and checking to see which predicting class at the end of the forward pass has the highest probability. Following this prediction, we compute the accuracy of the predicted labels in the function `accuracy.m`. In this function, we calculate the ratio of number of samples predicted correctly to the total number of samples tested. Now that we have evaluated the accuracy of our model, it is helpful to visualize it graphically. To do this, we use the function `visualize_history`. This function plots the training loss and testing accuracy versus time under the defined hyperparameters.

III. Discussing Changes in Hyperparameters

A. Default Hyperparameters

Below, Figure 2 displays the results when using the default parameters for this problem of 150 training epochs, a learning rate of 0.01, and the number of layers set to 2.

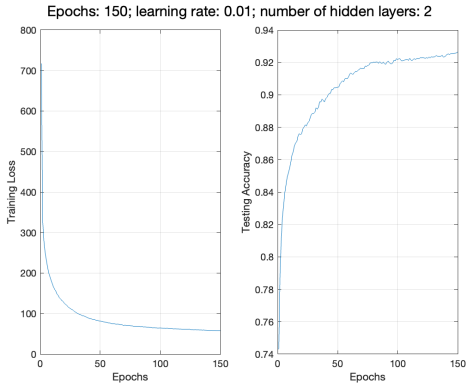

Fig. 2. Training Progress for Default Parameters

B. Varying Number of Epochs

We want to analyze the effect of changing the hyperparameters associated with training this neural network. To start we vary the epoch, setting it equal to 50, 150, and 300 and run the training and test individually for each of these three cases. For these cases, we keep the learning rate and number of layers at their default values of 0.01 and 2, respectively. The results are shown in Figures 3-5.

As intuitively expected, the time it took to train and test the neural network increased as the number of epochs increased. On my machine, it took 35 seconds to run with 50 epochs, 103 seconds to run with 150 epochs, and 205 seconds with 300 epochs. Looking at the output training loss when the number of epochs is 50 (figure 2), the training loss begins to plateau. However, when comparing this to when the number of epochs is 150, there is a clear decrease in training loss. However, the results for training loss are similar between when the number of epochs is 150 and when the number of epochs is 300. Although the training loss is less when the number of epochs is 300, there becomes a point where increasing the number of epochs is not with the increase in computation time. Now, looking at the testing accuracy, there is a very sharp and immediate increase until the number of epochs causes the testing accuracy to plateau. For the case where the number of epochs is 50, the data has not approached a more terminal value, but once the number of epochs is increased to 150, the data has approached a more constant testing accuracy, as compared to the results of when there are 300 epochs. The number of epochs needs to be chosen to be high enough such that the results begin to converge, but choosing it to be too high will make running the code to be impractical.

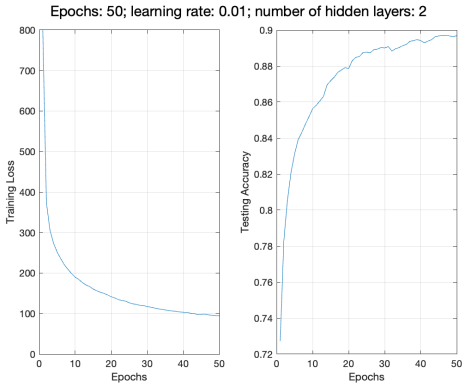

Fig. 3. Training Progress with Number of Epochs Set to 50

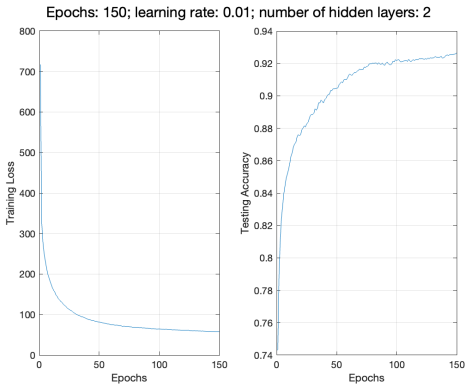

Fig. 4. Training Progress for Default Parameters (Number of Epochs Set to 150)

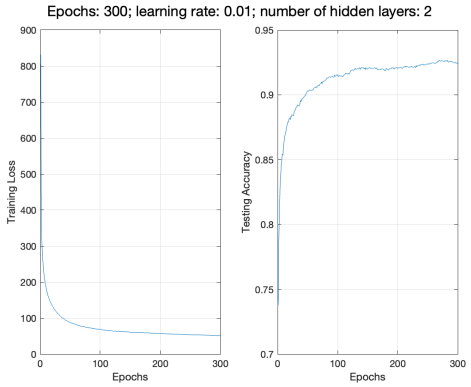

Fig. 5. Training Progress with Number of Epochs set to 300

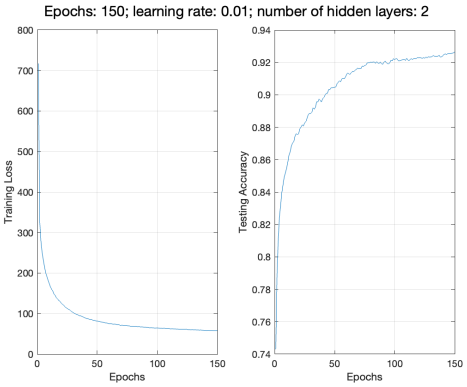

Fig. 7. Training Progress for Default Parameters (Learning Rate Set to 0.01)

C. Varying Learning Rate

Next, we vary the learning rate and the results are shown in Figures 6-8. We set the learning rate equal to 0.1 0.01, and 0.001 and run the training and test individually for each of these three cases. In doing this, we keep the number of epochs and number of layers constant at their default values of 150 and 2. The time it took to run each simulation was all 108 +/- 1 second, and thus the learning rate had no discernible effect on the runtime of the training and testing. Increasing the learning rate to 0.1 has a very noticeably negative effect, where although the results start to look promising at around 50 epochs, the results diverge and training loss increases and testing accuracy decreases as the number of epochs continue to increase. Decreasing the learning rate to 0.001 has the effect of making the data appear smoother, however the training loss still increases and the testing accuracy decreases, when compared to the default learning rate of 0.01. This goes to show that there is an ideal learning rate, where if it is made too small or too large, the training will not be as effective.

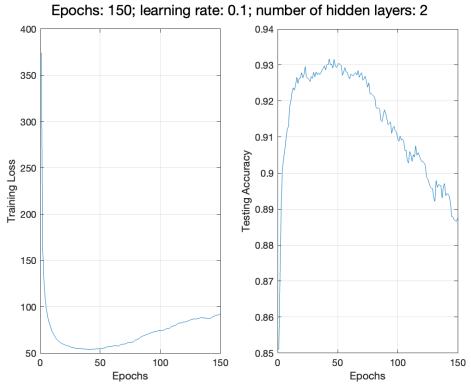

Fig. 6. Training Progress with Learning Rate Set to 0.1

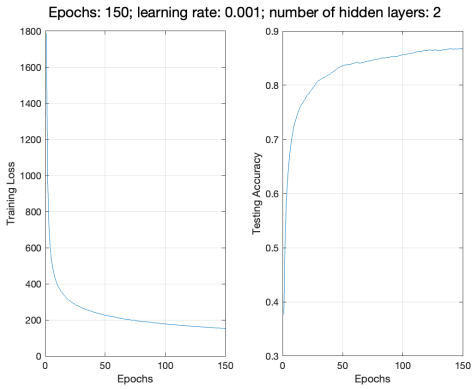

Fig. 8. Training Progress with Learning Rate Set to 0.001

D. Varying Number of Layers

Next, we vary the number of layers and the results are shown below in Figures 9-11. We use the prompted number of layers of 2, 3, and 5. Once again, we keep the other hyperparameters at their default values, with the number of epochs equal to 150, and the learning rate equal to 0.01. The runtime for each of these different parameters on my machine is 113, 123, and 155 seconds respectfully, showing that there is an increase in the computation time as the number of layers increases from 2 to 3, and from 3 to 5, however they are relatively small increases. When increasing the number of layers from 2 to 3, the data looks promising for approximately the first 50 epochs. However, after 50 epochs the training loss increases and the testing accuracy decreases. When increasing the number of nodes from 2 to 5, the training appears to be unreliable with a lot of noise. Although the training loss is overall trending downwards and the training accuracy is trending upwards, there is too much noise, making the result highly dependent on the number of epochs chosen. To summarize, increasing the number of hidden layers above 2 in this case will have a negative effect. The initially set default hyperparameters prove to yield the best results for this project.

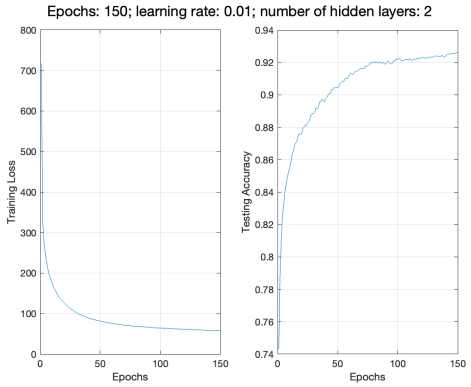

Fig. 9. Training Progress for Default Parameters (Number of Layers Set to 2)

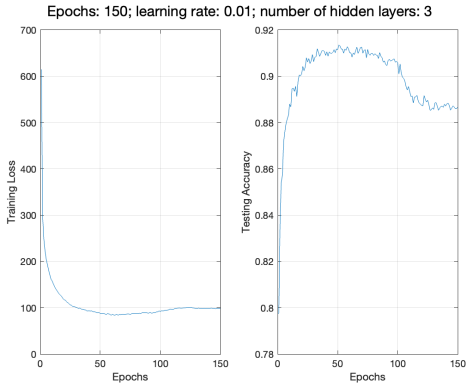

Fig. 10. Training Progress with the Number of Layers Set to 3

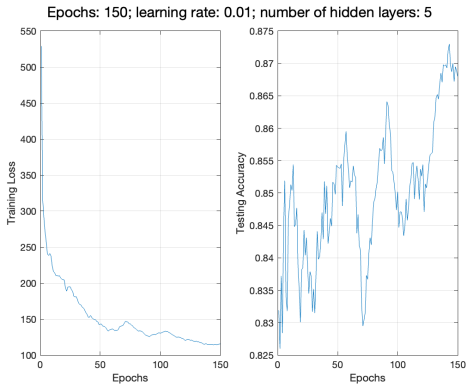

Fig. 11. Training Progress with the Number of Layers Set to 5

REFERENCES

[1] Jawed, Khalid. ”Lecture 11”, 8 Nov 2023
[2] Jawed, Khalid. ”Lecture 12”, 15 Nov 2023
[3] Jawed, Khalid. ”Homework 3, Fall 2023”