MAE 263F Homework #3

Atti W. Lau

I. Assignment Introduction

This assignment revolves around a neural network that is trained and used to recognize images of handwritten numbers. Specifically, a feedforward neural network is created for a classification problem where it classifies these images by creating a probability distribution for which number it believes an image represents. Each image is a square grayscale image that is 28 pixels in length and width.

The image is fed through the structure of the neural network by moving forward through its layers, hence the "feedforward" description. The output of the neural network is a vector of size K x 1, where K = the number of classes. In this case, K = 10 as it represents the digits 0 to 9. This output vector is the probability distribution generated by the neural network. The first element of the vector represents the probability predicted by the neural network that the image shows the number "0", the second element is the probability that the image shows "1", etc.

An overview of the layer structure of the neural network is shown below:

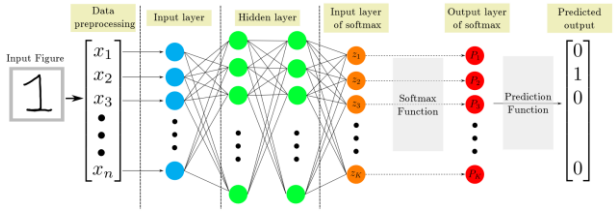

Figure 1: Layer structure of the neural network [1].

In this task, the neural network is first trained on a set of images accompanied by their correct labels, called the MNIST dataset.

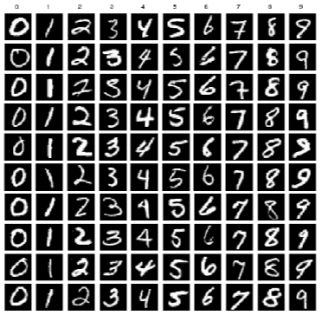

Figure 2: The MNIST dataset used to train the model [1].

The performance of the neural network is then assessed by feeding it images from a test set, then comparing its predictions with the correct labels. The performance of the neural network when various hyperparameters are varied is examined.

II. Neural Network Code Detail

The code works when the main function, *project_005621626*, is run on its own. It imports the datasets, structures the neural network and its hyperparameters, trains the network, tests the network, and displays plots of the neural network's training loss and accuracy with each epoch. The overall architecture of the main code is shown in figure 3.

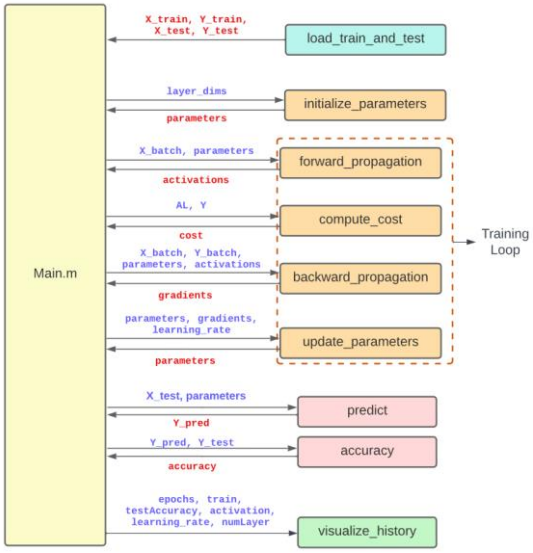

Figure 3: Architecture of main code [1].

A. Data Processing

At the beginning of the main function, after the clear and close all commands are given, the image and label data from both the training and test datasets are loaded in with the function *load_train_and_test_data()*. This constitutes the data preprocessing layer in the neural network structure as seen in Figure 1. In the data preprocessing layer, each input figure is "flattened" into a single-column vector with 28*28 = 784 elements, each element representing the intensity of a pixel from 0 to 1. All the flattened images in a dataset are stored in a 784 x N array, where N is the number of images in the dataset. Finally, *load_train_and_test_data()* returns four arrays: X_train, the array of flattened images for training; Y_train, the 10 x N array of labels for the training data; X_test, the flattened images for testing and Y_test, the corresponding labels for the testing images.

B. Defining Network Architecture

The next part of the main code defines the architecture of the neural network. First, the sizes of the input and output arrays are calculated from the preprocessed image and label datasets. Then the hyperparameters controlling the characteristics of the neural network are defined. These

include the number of neurons per hidden layer *neurons*, the number of hidden layers *numLayer*, the learning rate *lr*, and the number of epochs. The layers are then constructed from the hyperparameters.

C. Model Training

The main code calls the function *initialize_parameters()* to initialize the weights and biases of the neural network. It takes the array *layer_dims*, an array that stores the array sizes of the various layers in the neural network. Each neuron in the layers of the neural network take the inputted data and calculate a weighted sum with a bias, which is then outputted to the next layer. These weights and biases define the relationships among the neurons of the network.

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

In each hidden layer, Weights are initialized to random values, and biases are initialized to be at 0.

Next, the model enters the training loop. The number of times that the code runs through the training loop is defined by the number of epochs. The model is trained using mini-batch gradient descent.

The training data is shuffled to prevent biases in the model. First, the batch of inputs is run forward through the entire neural network structure during the forward pass in *forward_propagation.m*. At the end of the input and hidden layers, the nonlinear activation function σ is applied to improve finding the relationships between the input images and output labels. In this neural network, the activation function is the tanh function. At the output layer, the output is run through the softmax function that produces the probability distribution showing the neural network's label predictions for the image.

$$P_i \equiv \mathrm{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

The next step in the main function is to compare the output prediction to the correct labels in the training dataset, from which the cross-entropy loss, or cost, is calculated. This occurs in the function *compute_cost()*. The cross-entropy loss is defined by

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

A neural network is generally better at making correct predictions when the cross-entropy loss is minimized, which is where back propagation is useful.

In the backpropagation step, done by *backward_propagation()* in the main function, the error is fed backwards through the layers of the neural network to further optimize the weights and biases. Using chain rule, it computes the gradient of the error against the various parameters in the neural network. It uses the principle of gradient descent to find the parameters where error is minimized. As the gradient gives the direction of greatest increase or decrease in error when a parameter is varied, the principle of gradient descent is to use that direction to find and follow the steepest descent, eventually arriving at a loss minimum. The results from *backward_propagation()* are then used to update the parameters with *update_parameters()*.

D. Model Testing

At the end of each epoch, the cross-entropy loss after the backpropagation step is recalculated and stored. The accuracy of the neural network is also measured by feeding through a test dataset separate from that used for training, then comparing the predictions against the correct labels for the testing dataset in *accuracy()*. Accuracy is simply measured as the percentage of labels that the neural network predicted correctly, taking the label with the greatest probability in its outputs as the neural network's prediction.

Finally, plots are generated by *visualizeHistory()*. The two plots generated are training loss vs. epoch and accuracy vs. epoch.

III. Results

A. Varying Training Epochs

To study the effect of the number of epochs on the performance of the neural network, the learning rate and number of layers were set to 0.01 and 2, respectively, and the code was run 3 times with the number of epochs set to 50, 150, and 300.

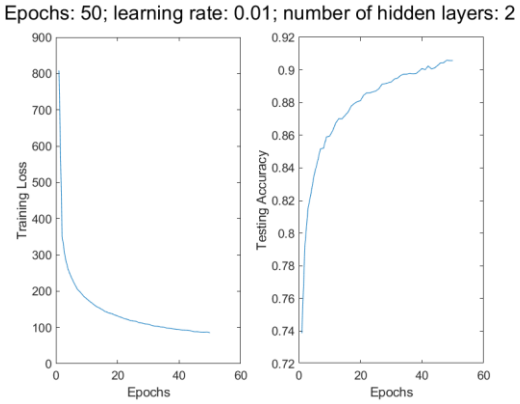

Figure 4: Loss and accuracy vs. epoch for 50 epochs.

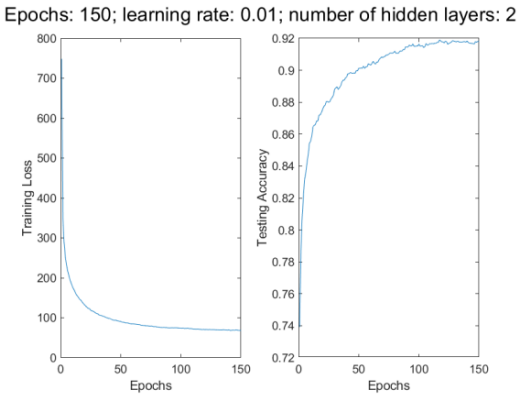

Figure 5: Loss and accuracy vs. epoch for 150 epochs.

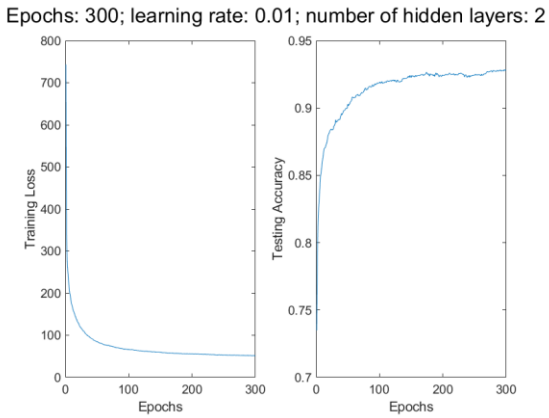

Figure 6: Loss and accuracy vs. epoch for 300 epochs.

The time taken by the entire program to run was directly related to the number of epochs, with more epochs causing it to take longer to finish. This is expected as the number of epochs is simply how many times the input batch is fed through to train the neural network.

Loss and accuracy followed similar trends for all three results. However, as the number of epochs increased, the loss and accuracy came closer to their lower and higher asymptotes, respectively. Therefore, a larger number of epochs generally increases the neural network's performance, albeit with diminishing returns when increasing epochs and with significant increases in training time.

B. Varying Learning Rates

The epochs and layers were kept at 150 and 2, respectively, and the learning rate was varied from 0.1, 0.01, and 0.001.

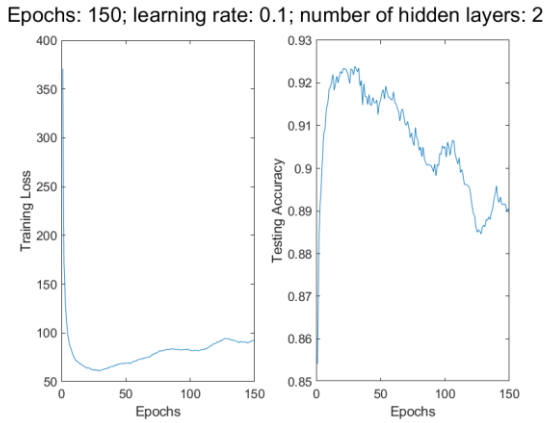

Figure 7: Loss and accuracy vs. epoch for 0.1 learning rate.

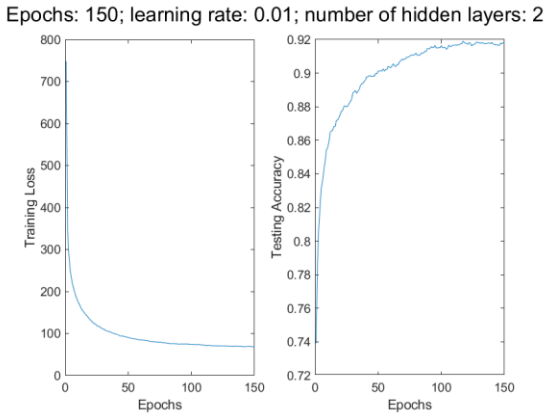

Figure 8: Loss and accuracy vs. epoch for 0.01 learning rate.

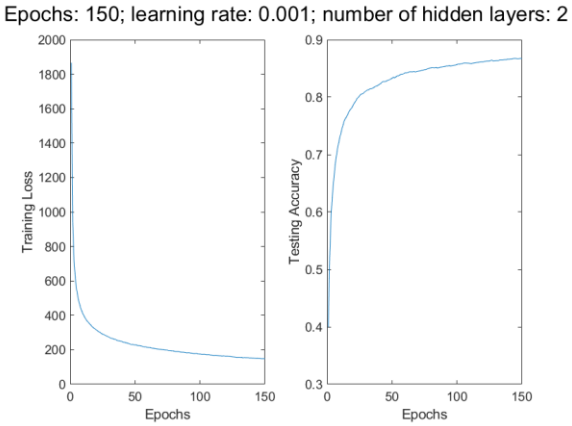

Figure 9: Loss and accuracy vs. epoch for 0.001 learning rate.

The time taken by the entire program to run was similar across the different learning rates. Notably, for the largest learning rate of 0.1, the neural network performed worse in general over epochs in the training phase, with the loss increasing and the accuracy decreasing, as well as making several jumps. This high learning rate seems to have caused the neural network to overshoot when calculating ideal parameters, resulting in poor performance. The slowest learning rate of 0.001 gave the smoothest progress in training loss and accuracy. However, the neural network was also prevented from improving quickly in the number of epochs that it was trained through, resulting in a lower performance than the intermediate learning rate of 0.01.

C. Varying the Number of Layers

The epochs and learning rate were set at 150 and 0.01, respectively, and the code was run 3 times with the number of layers set to 2, 3, and 5.

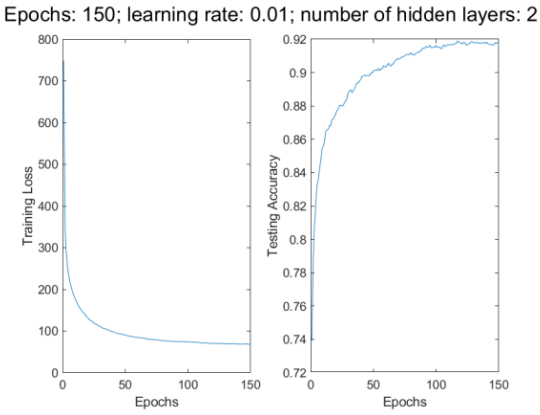

Figure 10: Loss and accuracy vs. epoch for 2 hidden layers.

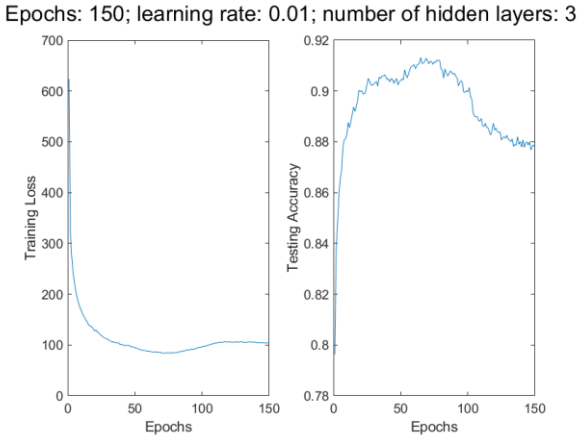

Figure 11: Loss and accuracy vs. epoch for 3 hidden layers.

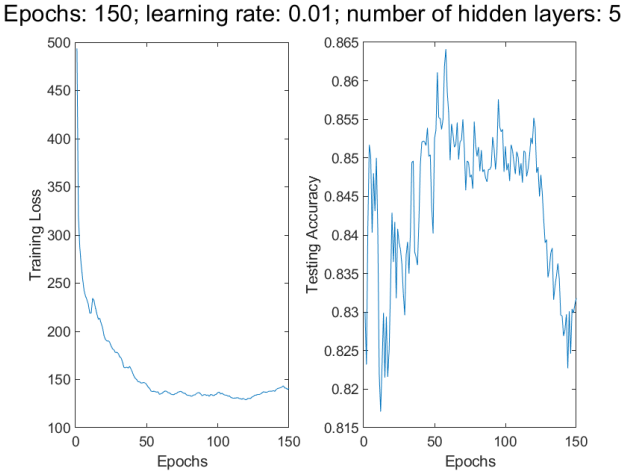

Figure 12: Loss and accuracy vs. epoch for 5 hidden layers.

The neural network with 2 hidden layers had the best performance, with performance generally decreasing with the number of layers. The network with 3 layers began with an increase in improvement like that with 2 layers but began to drop in performance around 75 epochs. The accuracy over the epochs was highly irregular in the neural network with 5 layers.

ACKNOWLEDGMENT

I thank Professor M. K. Jawed for his guidance and provided functions for completing this assignment.

REFERENCES

[1] M. K. Jawed, “Homework 3” . UCLA, 2023.