Deep Learning PA2 Report

Sahith Bhamidipati

March 2022

1 Problem

The purpose of this assignment is to be able to classify images of digits ranging from 0-9. The program that forms a solution to this problem is expected to classify these images with a reasonably high accuracy, such as in the 90%'s.

2 Model

This model is multi-class classification model that employs a neural network to classify images of digits as the correct digit. The model takes in images of digits from 0-9 as input and returns classifications for each of the images as output.

The model consists of one input layer of size 784×1, which is the number of pixels of the input images, two hidden layers of size 100×1 each, and one output layer of size 10×1 for the number of possible classifications of an image. This means that there are three total weight and bias matrices: one for each of the hidden layers and one for the output layer.

The training of the model begins with forward propagation. In forward propagation, the first hidden layer is computed using

$$H^1 = \phi((W^1)^T X + W_0^1)$$

where $\phi(x)$ is the ReLU activation function $\text{ReLU}(x) = max(0, x)$ and X is the input vector of a single sample of data.
The next hidden layer is computed using

$$H^2 = \phi((W^2)^T H^1 + W_0^2)$$

using the same activation function.
Lastly, the output layer is computed using

$$H^3 = \hat{y} = \sigma((W^3)^T H^2 + W_0^3)$$

where $\sigma(\vec{x})$ is the softmax output function

$$\sigma(\vec{x}) = \begin{bmatrix} \frac{e^{x_0}}{\sum_j^N e^{x_j}} & \cdots & \frac{e^{x_N}}{\sum_j^N e^{x_j}} \end{bmatrix}^T$$

and $\hat{y}$ is the model's classification of the input vector.

Next, back propagation to compute each gradient for the layers of the model is done in mini-batches of size 50. First, the output gradient is computed using

$$\nabla \hat{y} = [-\frac{1}{M}(y - \hat{y})]^{[K \times M]}$$

where K is the number of possible classes (10), M is the size of the mini-batches, y represents the true labels for each of the 50 samples in the batch, and $\hat{y}$ contains the output for each of the 50 samples in the batch.
Next, the output layer weight gradients and the gradient of the second hidden layer are calculated using

$$\nabla W_0^3 = \nabla \hat{y} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}^{[M \times 1]} = \begin{bmatrix} \sum_{i=1}^M \nabla \hat{y}[1] \\ \vdots \\ \sum_{i=1}^M \nabla \hat{y}[K] \end{bmatrix}^{[K \times 1]}$$

$$\nabla W^3 = [H^2 (\nabla \hat{y})^T]^{[N_L \times K]}$$

$$\nabla H^2 = [W^3 \nabla \hat{y}]^{[N_L \times M]}$$

where $N_L = 100$ is the size of the hidden layers.
Next, the gradients of the remaining hidden layer are computed using

$$\nabla W_0^2 = \nabla H^2 \frac{\partial \phi(H^2)}{\partial H^2} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}^{[M \times 1]} = \begin{bmatrix} \sum_{i=1}^M (\nabla H^2 \frac{\partial \phi(H^2)}{\partial H^2})[1] \\ \vdots \\ \sum_{i=1}^M (\nabla H^2 \frac{\partial \phi(H^2)}{\partial H^2})[N_L] \end{bmatrix}^{[N_L \times 1]}$$

$$\nabla W^2 = [H^1 (\nabla H^2 \frac{\partial \phi(H^2)}{\partial H^2})^T]^{[N_L \times N_L]}$$

$$\nabla H^1 = [W^2 (\nabla H^2 \frac{\partial \phi(H^2)}{\partial H^2})]^{[N_L \times M]}$$

where the partial derivative of the ReLU function $\phi(x)$ is 1 if $x > 0$ else 0.
Lastly, the gradient of the input weights are computed using

$$\nabla W_0^1 = \nabla H^1 \frac{\partial \phi(H^1)}{\partial H^1} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}^{[M \times 1]} = \begin{bmatrix} \sum_{i=1}^M (\nabla H^1 \frac{\partial \phi(H^1)}{\partial H^1})[1] \\ \vdots \\ \sum_{i=1}^M (\nabla H^1 \frac{\partial \phi(H^1)}{\partial H^1})[N_L] \end{bmatrix}^{[N_L \times 1]}$$

$$\nabla W^1 = [X (\nabla H^1 \frac{\partial \phi(H^1)}{\partial H^1})^T]^{[N \times N_L]}$$

where N is the size of the input data.

After the gradients are computed, each of the weights and biases can be updated for each layer using

$$W^l(t) = W^l(t-1) + \alpha(\nabla W^l + \lambda \frac{\partial R(W^l)}{\partial W^l})$$

where α is the learning rate, λ is the regularization rate, and

$$\frac{\partial R(W^l)}{\partial W^l} = sign(W^l)$$

is the L1 regularization of W^l.

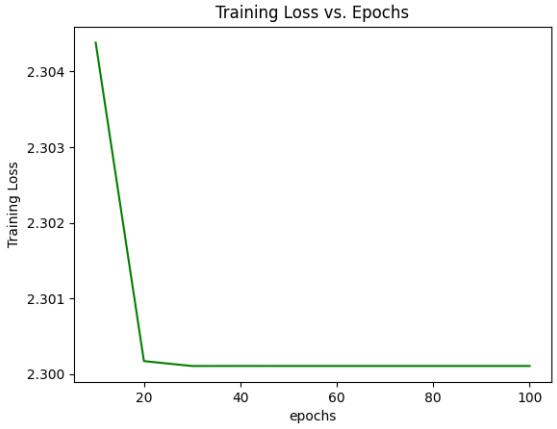
Lastly, the loss can be calculated on the new weights to ensure the model is training towards a loss of 0. The average loss over a mini-batch is computed by doing forward propagation on the same batch of data, then calculating

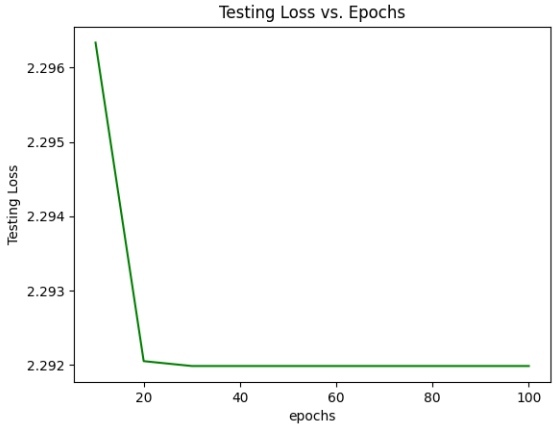
$$\frac{1}{M} \sum_{m=1}^{M} \sum_{k=1}^{K} -y[m][k] \log \hat{y}[m][k]$$

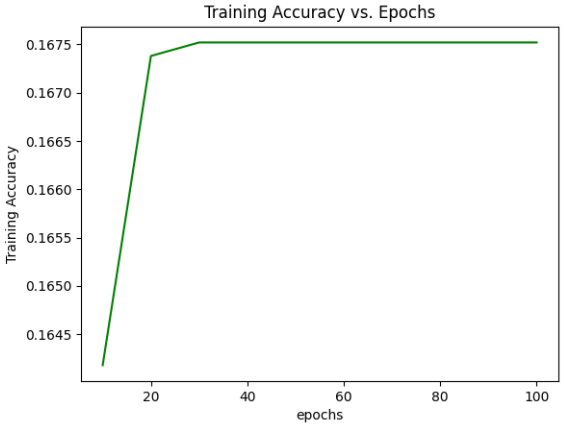
on the output of the forward propagation. Once the entire process is repeated for each mini-batch in the overall training set of data, one epoch of training is complete. This process is repeated until the model reaches convergence, meaning that its training is complete.

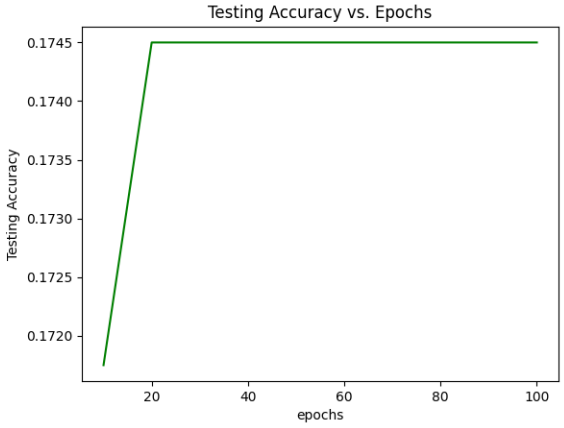
3 Plot

Plots for the training and testing loss, training and testing accuracy, and training and testing classification error for each digit are shown below:

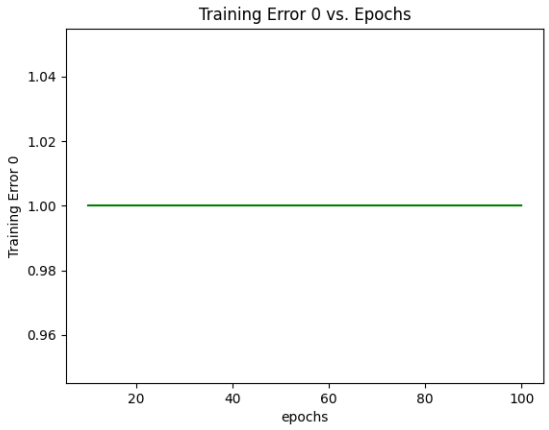


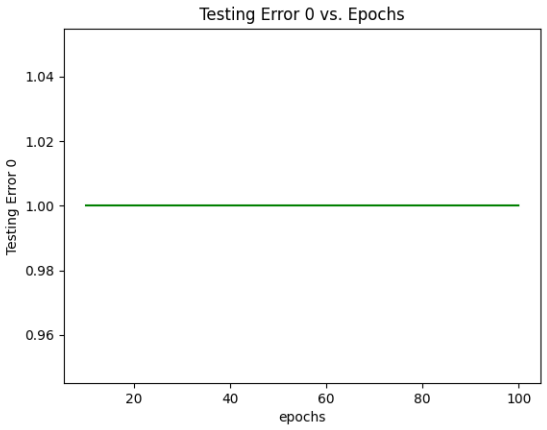




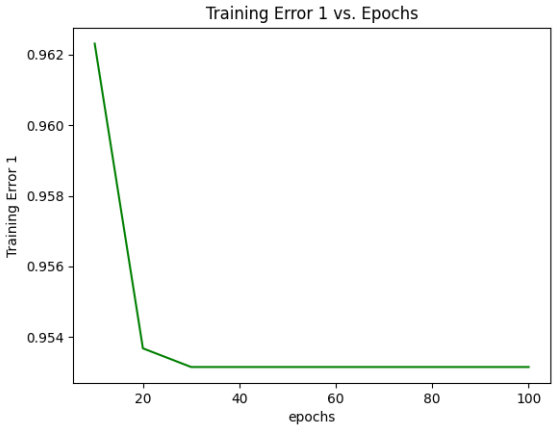


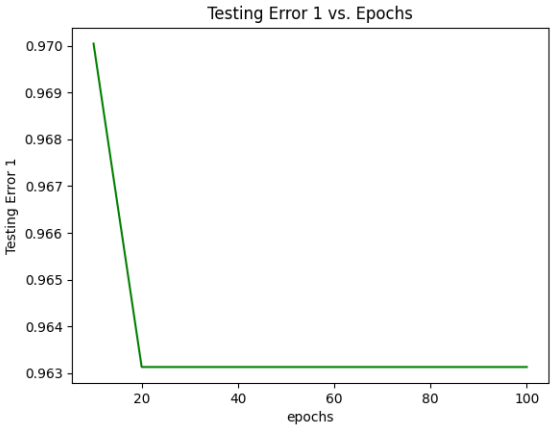
Digit 0:



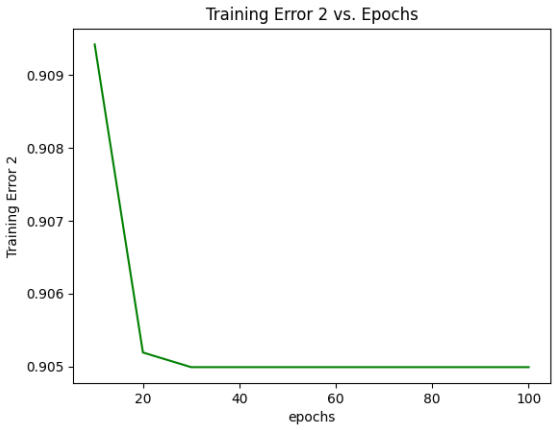


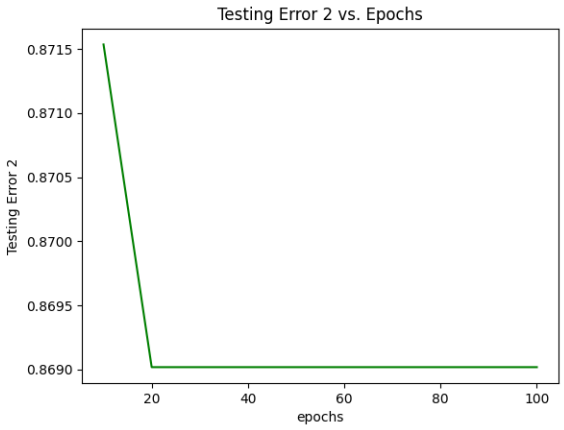
Digit 1:



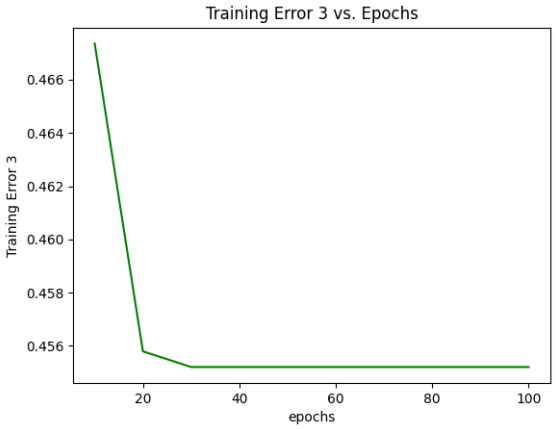


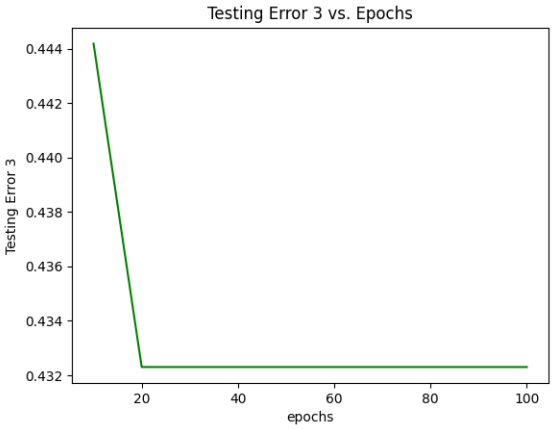
Digit 2:



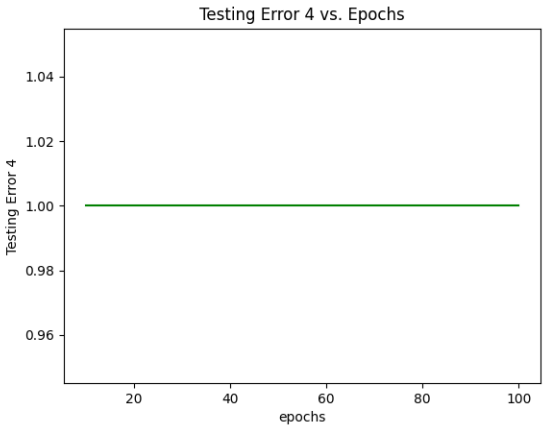


Digit 3:

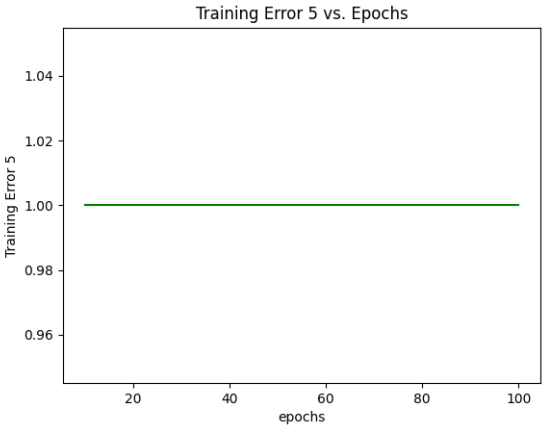


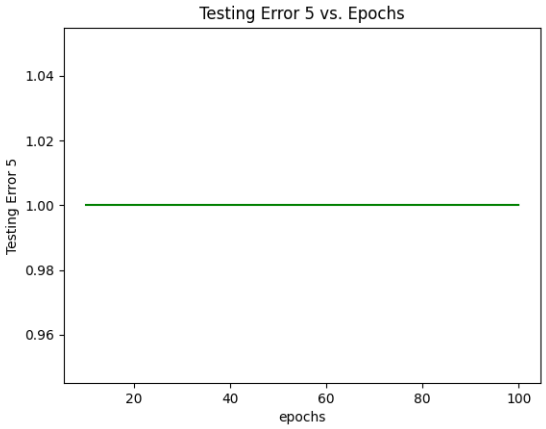


Digit 4:

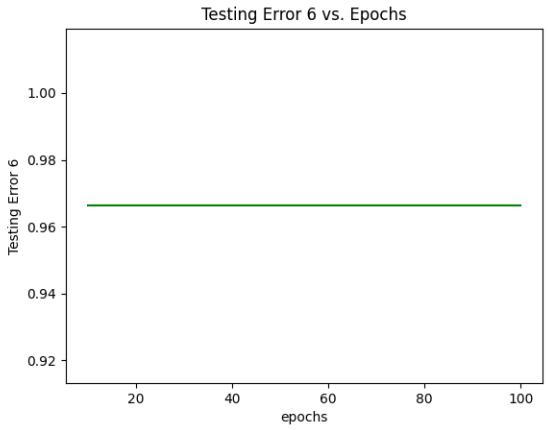


Digit 5:

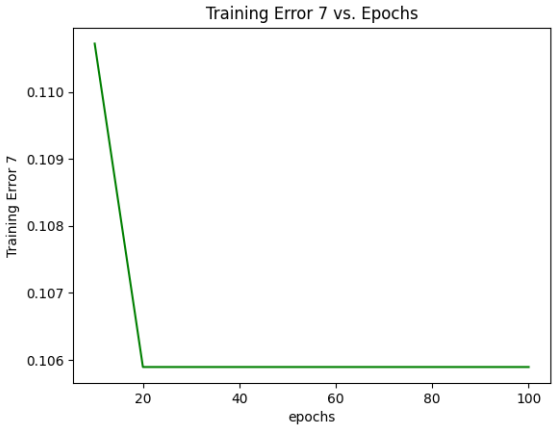


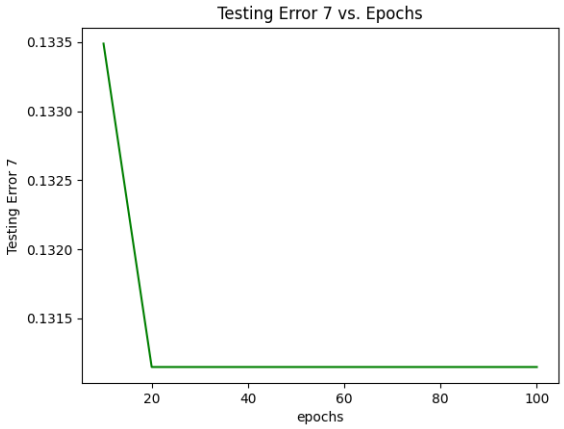


Digit 6:

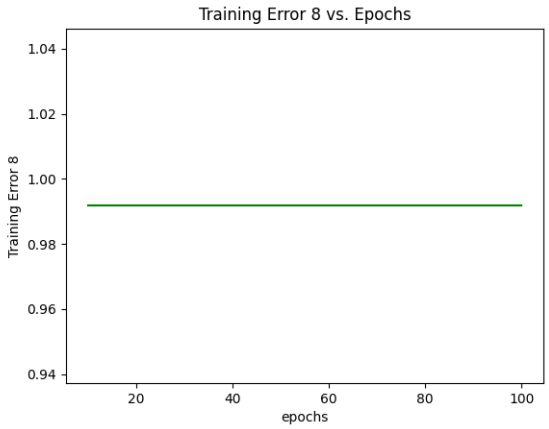


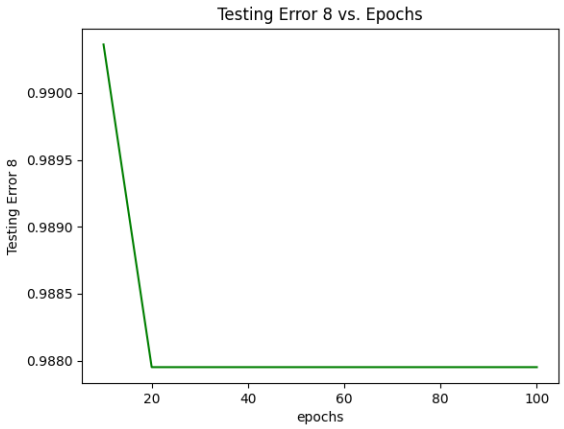
Digit 7:



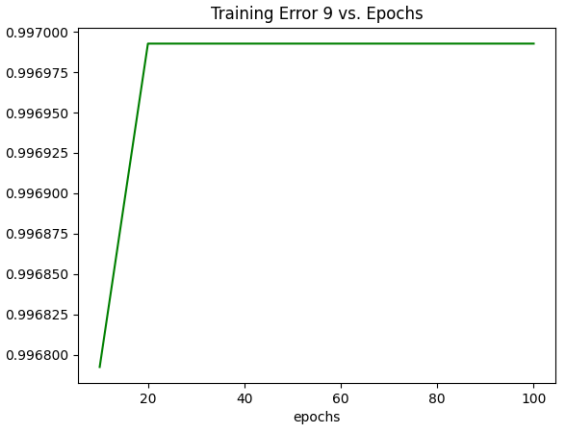


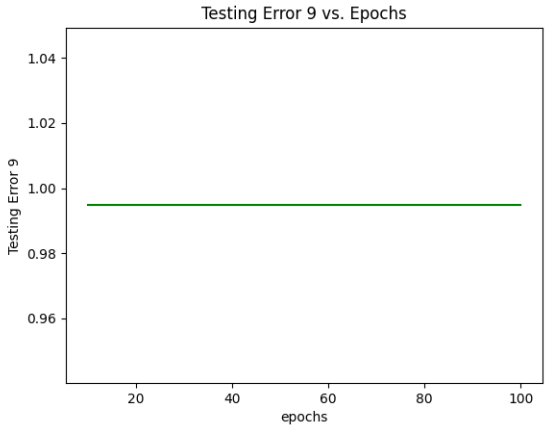
Digit 8:





Digit 9:





The final average classification error of the run was 82.55%

4 Discussion

This model was built with various python modules, including Numpy 1.22.2 and Tensorflow 2.8.0 for all computations. The model was run on an Apple M1 chip on a 2021 Macbook Pro running macOS Monterey. The M1 chip is claimed to be optimized for machine learning, according to Apple. The optimized nature of Tensorflow sped up the training of the model greatly as opposed to relying more heavily on pure Python.

The hyperparameters chosen were based running a series of tests on various hyperparameter values and choosing the value that produces the best accuracy. A learning rate too large would make the model fail to converge due to the weights changing at too large of a rate to reach a minimum, while a value too small would prevent substantial training from occurring because of the little impact the gradients will have on the weight updates. At first, the learning rate was set to $\alpha = 0.001$. This seemed to work for some iterations, but quickly failed in later iterations causing the loss to increase rapidly. To combat this, a formula for learning rate decay was put into effect. This reduced the rate from an initial value of $\alpha = 0.001$ by half every 2 iterations. With this decay in place, the model performed much better without break in the middle of training.
For the regularization rate, the value could be relatively small due to the large size of the training dataset, which would decrease the likeliness of overfitting even without regularization. In the end, the best regularization rate was $\lambda = 0.001$.

The weight initialization was another key variable aspect of the model. Having a faulty initialization could break the training entirely. At first, the weights were initialized using numpy's random.uniform function with a low range of 0 and a high limit of 0.1. This seemed to be less robust than numpy's random.normal function, however, which provides values on a normal distribution instead of uniform. For this reason, the weights are initialized using this distribution with a scale of 0.1.