ECSE 4850 Programming Assignment 2

Training a neural network with two hidden layers

Problem Statement

The objective of this programming assignment is to implement a neural network model with two hidden layers for the purpose of classifying handwritten digits 0-9. Only minimal usage of the Tensorflow machine learning library is allowed and therefore the functions for computing the loss of the model and the gradients of the loss function must be implemented from scratch. Additionally, the back propagation portion of the training loop must be implemented from scratch. When training, a random batch of 50 images must be selected from the training data set, output must be computed for each of these images, gradients must be computed for each layer using back propagation, and then the average of the gradients of the 50 images will be applied to the weights for each layer. While the model is training, performance data must be collected in the form of loss function value curves and average inaccuracy curves over iterations for the training batch and the testing set.

Model Description

The model for this assignment is a neural network structure with 4 fully connected layers total:

Layer #	Layer Name	Symbol	Nodes	Weight Matrix	Bias Vector	Activation
1	Input Layer	$\boldsymbol{X}$	784	-	-	-
2	Hidden Layer 1	$\boldsymbol{H}^1$	100	$\boldsymbol{W}^1$ (784×100)	$\boldsymbol{W}_0^1$ (100×1)	ReLU
3	Hidden Layer 2	$\boldsymbol{H}^2$	100	$\boldsymbol{W}^2$ (100×100)	$\boldsymbol{W}_0^2$ (100×1)	ReLU
4	Output Layer 2	$\hat{\boldsymbol{Y}}$	10	$\boldsymbol{W}^3$ (100×10)	$\boldsymbol{W}_0^3$ (10×1)	Softmax

Table 1: Model architecture summary

Environment Settings & Hyper Parameters

Environment

For development of this project, an Anaconda virtual environment was created with Python 3.8 and Tensorflow 2.3.0, and programming and debugging was completed using the Spyder IDE. To improve model training speed, Tensorflow was connected with my system's GPU via CUDA version 10.1 and CuDNN version 7.6. For reference, my system has an Intel Core i7-7700HQ 4-core CPU and an NVIDIA GeForce GTX 1050 GPU.

Hyper Parameters

- *Learning Rate*: For this assignment, an adaptive learning rate (η) was used. The learning rate started at $\eta^1 = 0.008$ and decreased with each epoch t according to the equation:

  $$\eta^t = \eta^1 \times \frac{1}{\sqrt{t}}$$

  For example, $\eta^1 = 0.008$, $\eta^2 = 0.00566$, $\eta^3 = 0.00462$, etc. Additionally, a different learning rate was used for each layer. Layer $\boldsymbol{H}^1$ was given a learning rate of 0.5η, layer $\boldsymbol{H}^2$ was given a learning rate of 1η, and layer $\hat{\boldsymbol{Y}}$ was given a learning rate of 2η. For example, for the first epoch, layer $\boldsymbol{H}^1$ has $\eta = 0.004$, layer $\boldsymbol{H}^2$ has $\eta = 0.008$, and layer $\hat{\boldsymbol{Y}}$ has $\eta = 0.016$.
- *Weight and Bias Initialization*: All weights were initialized to random values from the distribution $N(0, 0.1)$ and all biases were initialized to 0.
- *Stopping Criteria*: The training was set to stop if the testing inaccuracy ratio dropped below 0.07 or if the maximum number of epochs was reached (set to 8). This epoch cap was placed to ensure that the model did not run forever in the event that an inaccuracy ratio that low could not be reached.

Pseudocode

```
    import training and testing data sets
    reshape data to vector format and normalize
    create keras layers
    initialize weights to random values in N(0, 0.1) distribution, initialize biases to 0

    while True
        randomly create 1000 batches of 50 images (randomness seeded by epoch count)
        for batch = 1 to 1000
            forward propagation to compute output for training batch and for the test set
            compute training loss and inaccuracy
            compute testing loss and inaccuracy

            if testing inaccuracy <= 0.07 or epoch == 8
                get weights from layers
                return weights and performance data
            end if

            get weights from layers
            back propagation to compute gradients for each layer's weights
            compute average gradients over the 50 images in the batch
            update weights for each layer
        end loop

        increment epoch counter
        update learning rate
    end loop

   save weights and plot performance data
```

Loss Function & Gradient Equations

To learn the parameters for the model, the cross-entropy of the output nodes is minimized. Given training data $\boldsymbol{D} = \{\boldsymbol{X}[m], \boldsymbol{Y}[m]\}$, $m = 1, 2, ...M$, where $\boldsymbol{Y}[m]$ is a 1-of-k encoded vector, the loss function is:

$$l(\boldsymbol{Y}[m], \hat{\boldsymbol{Y}}[m]) = -\sum_{k=1}^{10} \boldsymbol{Y}[m][k] \times \log(\hat{\boldsymbol{Y}}[m][k]) \tag{1}$$

And the gradient of the loss function is:

$$\nabla\hat{\boldsymbol{Y}}[m] = -\frac{\partial l(\boldsymbol{Y}[m], \hat{\boldsymbol{Y}}[m])}{\partial \hat{\boldsymbol{Y}}} = \begin{bmatrix} \frac{\boldsymbol{Y}[m][1]}{\hat{\boldsymbol{Y}}[m][1]} \\ \vdots \\ \frac{\boldsymbol{Y}[m][10]}{\hat{\boldsymbol{Y}}[m][10]} \end{bmatrix} \tag{2}$$

Using Equation 2, we can begin the back propagation process for computing the gradients of each layer of the neural network. The first step is to use the gradient chain rule to compute the gradients for the last layer: $\nabla\boldsymbol{W}^3$, $\nabla\boldsymbol{W}_0^3$, and $\nabla\boldsymbol{H}^2$. For ease of notation, we will denote $\boldsymbol{z}[m] = (\boldsymbol{W}^3)^T\boldsymbol{H}^2[m] + \boldsymbol{W}_0^3$ and $\boldsymbol{z}[m][k] = (\boldsymbol{W}_k^3)^T\boldsymbol{H}^2[m] + \boldsymbol{W}_0^3[k]$. Note that σ_M here denotes the softmax function.

$$\begin{aligned} \nabla\boldsymbol{W}^3[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}^3}\frac{\partial \sigma_M(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]}\nabla\hat{\boldsymbol{Y}}[m] \\ \nabla\boldsymbol{W}_0^3[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}_0^3}\frac{\partial \sigma_M(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]}\nabla\hat{\boldsymbol{Y}}[m] = \frac{\partial \sigma_M(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]}\nabla\hat{\boldsymbol{Y}}[m] \\ \nabla\boldsymbol{H}^2 &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{H}^2}\frac{\partial \sigma_M(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]}\nabla\hat{\boldsymbol{Y}}[m] = \boldsymbol{W}^3\frac{\partial \sigma_M(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]}\nabla\hat{\boldsymbol{Y}}[m] \end{aligned} \tag{3}$$

where:

$$\begin{aligned}
\frac{\partial \sigma_M(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} &= \begin{bmatrix} \frac{\partial \sigma_M(\boldsymbol{z}[m][1])}{\partial \boldsymbol{z}[m]} & \dots & \frac{\partial \sigma_M(\boldsymbol{z}[m][10])}{\partial \boldsymbol{z}[m]} \end{bmatrix} \\
\text{where } \frac{\partial \sigma_M(\boldsymbol{z}[m][k])}{\partial \boldsymbol{z}[m]} &= \begin{bmatrix} \frac{\partial \sigma_M(\boldsymbol{z}[m][k])}{\partial \boldsymbol{z}[m][1]} \\ \vdots \\ \frac{\partial \sigma_M(\boldsymbol{z}[m][k])}{\partial \boldsymbol{z}[m][10]} \end{bmatrix} \\
\text{where } \frac{\partial \sigma_M(\boldsymbol{z}[m][k])}{\partial \boldsymbol{z}[m][k']} &= \begin{cases} \sigma_M(\boldsymbol{z}[m][k'])(1-\sigma_M(\boldsymbol{z}[m][k'])) & \text{if } k'=k \\ -\sigma_M(\boldsymbol{z}[m][k])\sigma_M(\boldsymbol{z}[m][k']) & \text{else} \end{cases}
\end{aligned} \tag{4}$$

and

$$\begin{aligned}
\frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}^3} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}[m][1]}{\partial \boldsymbol{W}^3} & \dots & \frac{\partial \boldsymbol{z}[m][10]}{\partial \boldsymbol{W}^3} \end{bmatrix} \\
\text{where } \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^3} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^3[\ ][1]} & \dots & \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^3[\ ][10]} \end{bmatrix} \\
\text{where } \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^3[\ ][j]} &= \begin{cases} \boldsymbol{H}^2[m] & \text{if } i=j \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{5}$$

Using the gradients obtained above, we can continue the back propagation process by computing the gradients for the second hidden layer: $\nabla \boldsymbol{W}^2$, $\nabla \boldsymbol{W}_0^2$, and $\nabla \boldsymbol{H}^1$. For ease of notation, we will denote $\boldsymbol{z}[m] = (\boldsymbol{W}^2)^T \boldsymbol{H}^1[m] + \boldsymbol{W}_0^2$ and $\boldsymbol{z}[m][k] = (\boldsymbol{W}_k^2)^T \boldsymbol{H}^1[m] + \boldsymbol{W}_0^2[k]$. Note that ϕ here denotes the ReLU function.

$$\begin{aligned}
\nabla \boldsymbol{W}^2[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}^2} \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^2[m] \\
\nabla \boldsymbol{W}_0^2[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}_0^2} \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^2[m] = \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^2[m] \\
\nabla \boldsymbol{H}^1[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{H}^1} \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^2[m] = \boldsymbol{W}^2 \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^2[m]
\end{aligned} \tag{6}$$

where:

$$\begin{aligned}
\frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} &= \begin{bmatrix} \frac{\partial \phi(\boldsymbol{z}[m][1])}{\partial \boldsymbol{z}[m]} & \dots & \frac{\partial \phi(\boldsymbol{z}[m][100])}{\partial \boldsymbol{z}[m]} \end{bmatrix} \\
\text{where } \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m]} &= \begin{bmatrix} \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m][1]} \\ \vdots \\ \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m][100]} \end{bmatrix} \\
\text{where} \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m][j]} &= \begin{cases} 1 & \text{if } i=j \text{ and } \boldsymbol{z}[m][i] > 0 \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{7}$$

and

$$\begin{aligned}
\frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}^2} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}[m][1]}{\partial \boldsymbol{W}^2} & \dots & \frac{\partial \boldsymbol{z}[m][100]}{\partial \boldsymbol{W}^2} \end{bmatrix} \\
\text{where } \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^2} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^2[\ ][1]} & \dots & \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^2[\ ][100]} \end{bmatrix} \\
\text{where } \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^2[\ ][j]} &= \begin{cases} \boldsymbol{H}^1[m] & \text{if } i=j \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{8}$$

Again using the gradients obtained above, we can continue (and complete) the back propagation process by computing the gradients for the first hidden layer: $\nabla \boldsymbol{W}^1$ and $\nabla \boldsymbol{W}_0^1$. For ease of notation, we will denote $\boldsymbol{z}[m] = (\boldsymbol{W}^1)^T \boldsymbol{X}[m] + \boldsymbol{W}_0^1$ and $\boldsymbol{z}[m][k] = (\boldsymbol{W}_k^1)^T \boldsymbol{X}[m] + \boldsymbol{W}_0^1[k]$. Note that ϕ here denotes the ReLU function.

$$\begin{aligned}
\nabla \boldsymbol{W}^1[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}^1} \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^1[m] \\
\nabla \boldsymbol{W}_0^1[m] &= \frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}_0^1} \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^1[m] = \frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} \nabla \boldsymbol{H}^1[m]
\end{aligned} \tag{9}$$

where:

$$\begin{aligned}\frac{\partial \phi(\boldsymbol{z}[m])}{\partial \boldsymbol{z}[m]} &= \begin{bmatrix} \frac{\partial \phi(\boldsymbol{z}[m][1])}{\partial \boldsymbol{z}[m]} & ... & \frac{\partial \phi(\boldsymbol{z}[m][100])}{\partial \boldsymbol{z}[m]} \end{bmatrix} \\ \text{where } \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m]} &= \begin{bmatrix} \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m][1]} \\ \vdots \\ \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m][100]} \end{bmatrix} \\ \text{where} \frac{\partial \phi(\boldsymbol{z}[m][i])}{\partial \boldsymbol{z}[m][j]} &= \begin{cases} 1 & \text{if } i = j \text{ and } \boldsymbol{z}[m][i] > 0 \\ 0 & \text{else} \end{cases}\end{aligned} \tag{10}$$

and

$$\begin{aligned}\frac{\partial \boldsymbol{z}[m]}{\partial \boldsymbol{W}^1} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}[m][1]}{\partial \boldsymbol{W}^1} & ... & \frac{\partial \boldsymbol{z}[m][100]}{\partial \boldsymbol{W}^1} \end{bmatrix} \\ \text{where } \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^1} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^1[\ ][1]} & ... & \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^1[\ ][100]} \end{bmatrix} \\ \text{where } \frac{\partial \boldsymbol{z}[m][i]}{\partial \boldsymbol{W}^1[\ ][j]} &= \begin{cases} \boldsymbol{X}[m] & \text{if } i = j \\ 0 & \text{else} \end{cases}\end{aligned} \tag{11}$$

Once all of the gradients have been computed, we must take the average of each weight and bias gradient over the 50 samples in each mini-batch. For $i = \{1, 2, 3\}$, we compute

$$\begin{aligned}\nabla \overline{\boldsymbol{W}}^i &= \frac{1}{50} \sum_{m=1}^{50} \nabla \boldsymbol{W}^i[m] \\ \nabla \overline{\boldsymbol{W}}_0^i &= \frac{1}{50} \sum_{m=1}^{50} \nabla \boldsymbol{W}_0^i[m]\end{aligned} \tag{12}$$

We can then update the weights for each layer using the learning rate η as described in the **Hyper Parameters** section. For $i = \{1, 2, 3\}$, we update

$$\begin{aligned}\boldsymbol{W}^i[t] &= \boldsymbol{W}^i[t-1] - \eta(\nabla \overline{\boldsymbol{W}}^i) \\ \boldsymbol{W}_0^i[t] &= \boldsymbol{W}_0^i[t-1] - \eta(\nabla \overline{\boldsymbol{W}}_0^i)\end{aligned} \tag{13}$$

Model Performance

During and after training, the performance of the model in terms of loss function value and classification accuracy was evaluated. The loss of the model over each training iteration is shown in Figure 1. (Note: for the entirety of the discussion of model performance, the term *iterations* refers to each mini batch of 50 images that the model trained on. The term *epochs* refers to the number of times the model trained on all 50,000 images in the training set. For example, each *epoch* consists of 1,000 *iterations*.)

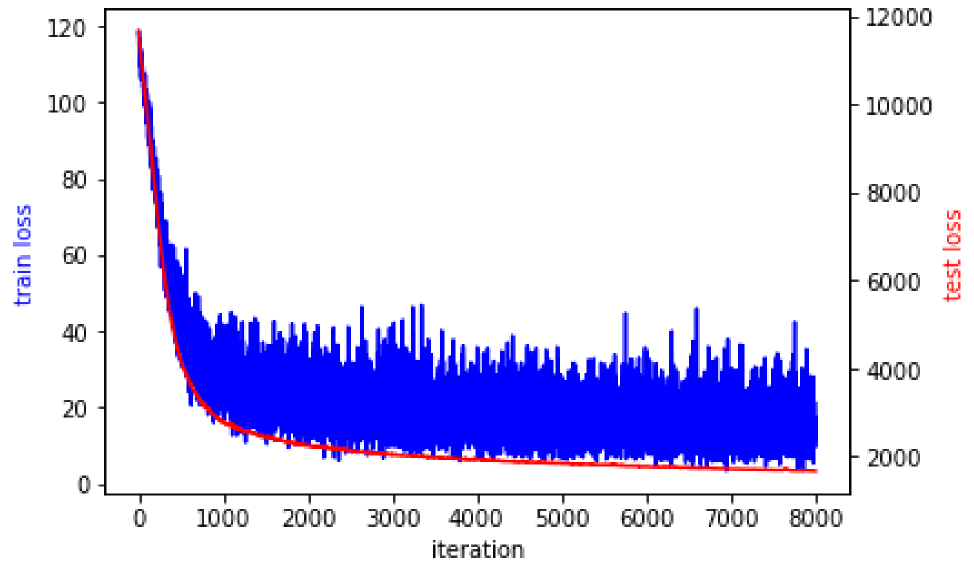


Figure 1: Model loss plotted as a function of iterations

The above plot shows both the training and testing loss of the model as a function of the number of iterations elapsed. As is evident from the plot, both the training and testing loss started at a large value and decreased quickly initially, eventually leveling out and decreasing more slowly. The testing loss tracks the training loss very closely, which is a good indication that the model is not overfitting. Additionally, we can see that the training loss function varies greatly from iteration to iteration; this is a result of the random batch of 50 images that are selected. Despite the variations, we can still see that, over time, the loss function is trending downwards. At iteration 8000, the training loss was 11.51 and the testing loss was 1675.58.

In addition to the loss values, the classification accuracy for each class was also evaluated after the completion of each iteration. The final classification inaccuracy ratio for each class, as well as the overall average, is shown below in Table 1. The "inaccuracy" values in Table 2 and the plots in Figures 2 & 3 are the ratio of incorrectly classified images for a specific class to the total number of images that appeared for that class.

Class	0	1	2	3	4	5	6	7	8	9	Average
Testing Inaccuracy	0.04	0.03	0.13	0.013	0.08	0.17	0.06	0.09	0.12	0.13	0.098

Table 2: Final testing inaccuracy for each class and the overall average

From Table 2, it is clear that the model is relatively accurate when making classification predictions. It appears that the model has the most trouble with classifying the digit 5, but is very accurate when classifying the digit 1. To illustrate the learning process further, the average training and testing inaccuracy is plotted as a function of iterations in Figure 2 below.

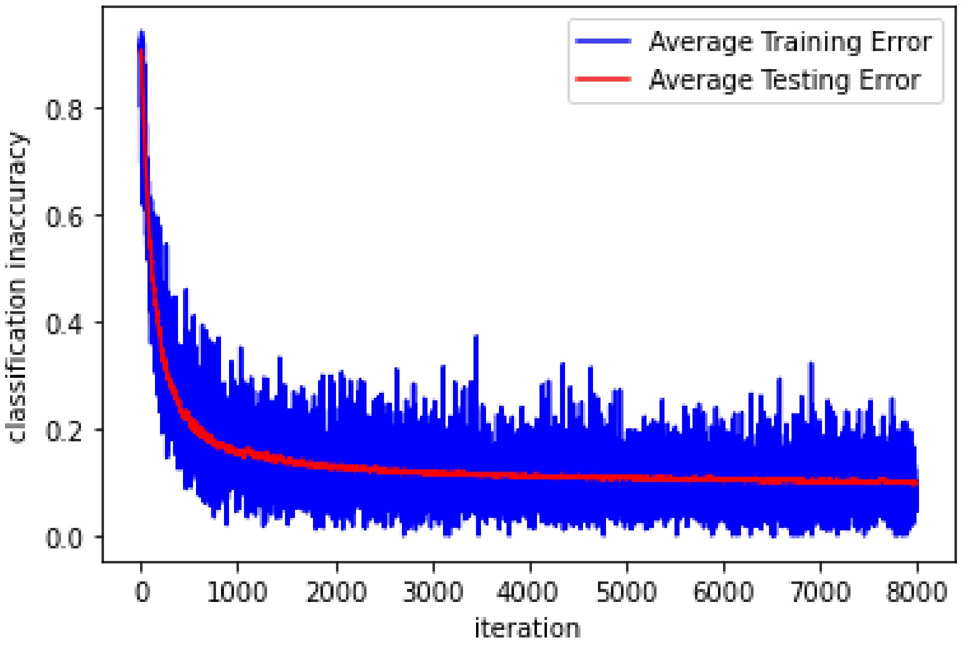
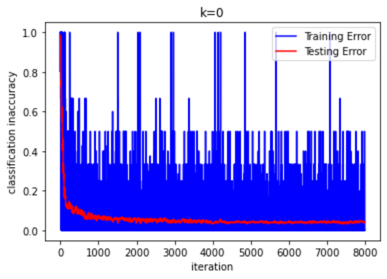


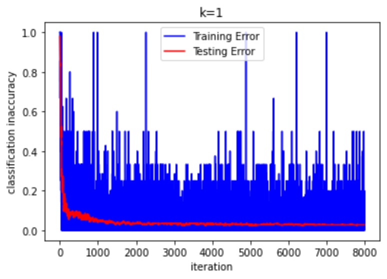
Figure 2: Average training and testing inaccuracy over time

Again, we can see that the average training and testing error follow each other very closely, showing that the model is not overfitting. We also see that the average training error varies greatly from iteration to iteration. This is a result of the mini batches being randomly selected sets of 50 images. If no images appear in the mini batch for a class, the inaccuracy of that class for that iteration defaults to 0. Similarly, if only one image appears in the mini batch for a class and the model classifies it incorrectly, the inaccuracy of that class for that iteration becomes 1. Therefore, there is great variation in the average training inaccuracy. Despite this, we can see that the average training error curve is trending downwards and becoming less varied as time goes on. Likewise, the average testing error curve decreased quickly at the start of training and then decreased more slowly, eventually starting to level off around 0.098 by the last 500 iterations or so. I believe that if I had let the model run for a bit longer, this error rate would decrease even more (in fact, it reached 0.095 inaccuracy a few iterations before the last iteration).

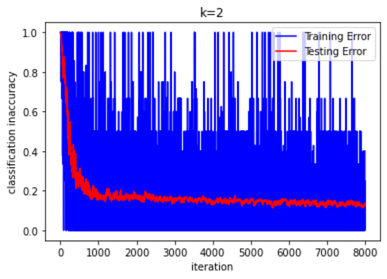
For reference, the training and testing inaccuracy for each class as a function of iterations is plotted in Figure 3 below. The training inaccuracy curves on these plots are more or less useless given the amount of variation in the plots (for the reasons described above), but the testing inaccuracy curves illustrate how the model performed while it learned. We can see, with the digit 1, how confident the model became with its ability to classify the digit. The curve dropped steeply and levelled off at a very low value. This is also the case for the digits 0 and 6. We can also see, with the digit 5, how much difficulty the model had with obtaining weights that resulted in accurate classifications. The curve dropped steeply at the beginning, started to decrease more slowly, but continued to zigzag up and down as time went on. This also occurred for the digits 4 and 9.



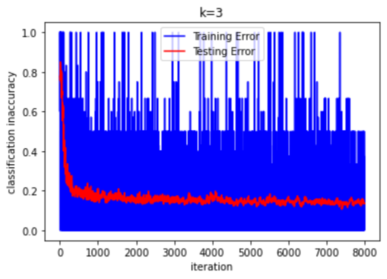
(a) Training and testing inaccuracy, k=0



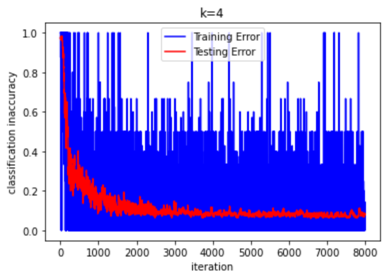
(b) Training and testing inaccuracy, k=1



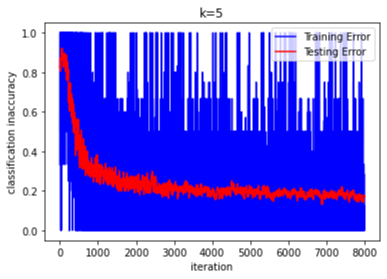
(c) Training and testing inaccuracy, k=2



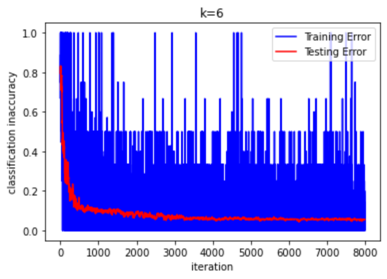
(d) Training and testing inaccuracy, k=3



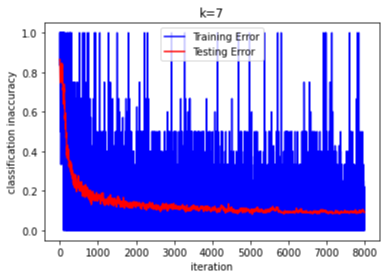
(e) Training and testing inaccuracy, k=4



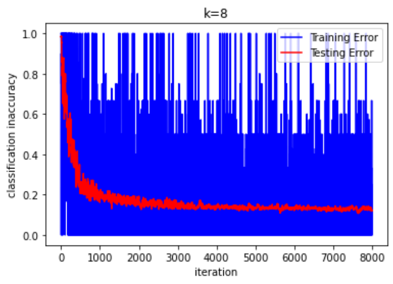
(f) Training and testing inaccuracy, k=5



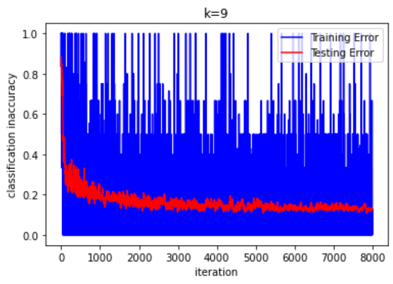
(g) Training and testing inaccuracy, k=6



(h) Training and testing inaccuracy, k=7



(i) Training and testing inaccuracy, k=8



(j) Training and testing inaccuracy, k=9

Figure 3: Training and testing accuracy for each class

Discussion

In conclusion, I believe I have succeeded in implementing the forward and back propagation steps for training a neural network to classify the digits 0-9. I was aiming to get the inaccuracy rate down to 0.07, but due to a lack of time (as a result of a lack of computing power), I had to tell the model to halt after only 8,000 iterations. This is also one of the reasons for my choice to use a learning rate of 0.008 as the starting learning rate for epoch 1. I wanted to set it to a large enough value that the model would converge quickly but small enough that the model wouldn't overfit or oscillate. I tried to run the model with a learning rate of 0.005 and 0.002, and while they started to converge, I had to stop them early because they would have taken far too long to reach the desired inaccuracy rate of 0.07. I also tried to run the model with a learning rate of 0.01, but it began to oscillate after only a few epochs and I stopped it under the impression that the oscillations would continue.

Future Considerations

If I were to complete this assignment again in the future, I would make a few changes. First, if I had the computing power and time, I would lower the learning rate to 0.002 or so in order to limit the amount of oscillation happening in the model (this can be seen in the inaccuracy plots in Figure 3 for a few of the classes). I would also look more into adjusting the learning rate for each parameter. I implemented different learning rates for each layer since that seemed to be a good choice based on the papers published on the topic, but there is likely a more optimal way of setting different learning rates for each parameter.

Additionally, I would let the model run for a much longer period of time. I was able to achieve an inaccuracy rate of 0.098 in 8,000 iterations, but the performance of the model indicates that this value likely would have continued to decrease had I let the model run longer. I would probably aim for at least 10,000 iterations, but even more would be optimal (of course, I would have to keep a close eye on the model's performance to make sure the model didn't start to overfit).

Lastly, in the assignment, we were instructed to compute the training and testing loss and accuracy after each iteration (each parameters update). In the future, I would choose to not compute the testing loss and error after every iteration, but instead after every epoch. This would also help to significantly speed up the training process.