ECSE 4850 Programming Assignment 1

Training a multiclass logistic regressor using the gradient descent method

---

In this report, I will detail all of the information necessary to understand how I implemented this model. I will discuss the Python environment used, the loss function and loss function gradient equations used, the pseudocode for the algorithm that was implemented, the performance of the model, and lastly how certain settings and hyper-parameter values were chosen.

Environment Details

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.

Loss Function and Loss Function Gradient

The cross entropy loss function was used for this model, which is defined as follows. 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,

$$L(\boldsymbol{D} : \boldsymbol{\Theta}) = -\sum_{m=1}^{M}\sum_{k=1}^{K} \boldsymbol{Y}[m][k] \log \sigma_M(\boldsymbol{X}^T[m]\boldsymbol{\Theta}_k) \tag{1}$$

where σ_M is the softmax function,

$$\sigma_M(\boldsymbol{X}^T[m]\boldsymbol{\Theta}_k) = \frac{e^{\boldsymbol{X}^T[m]\boldsymbol{\Theta}_k}}{\sum_{k'=1}^{K} e^{\boldsymbol{X}^T[m]\boldsymbol{\Theta}'_k}} \tag{2}$$

The actual implementation of this loss function required a few changes to be made in order to guarantee numerical stability. As written, this function is prone to overflow (exponentials shooting up to infinity) and underflow (values becoming too small to be stored without the system approximating them to 0). To fix this, we can use the identity

$$\sigma_M(\boldsymbol{x}) = \sigma_M(\boldsymbol{x} + c), \quad c \in \mathbb{R} \tag{3}$$

Equation (3) allows us to subtract the maximum element of the inputted vector $\boldsymbol{x}$, guaranteeing that $\boldsymbol{x}$ has only non-positive elements and at least one zero element. This has the effect of ensuring that no exponential terms shoot off to infinity and that the denominator will always have a term equal to 1 (and therefore protecting against any "divide by zero" errors). The actual implementation in the program looks more like:

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{X}^T[m]\boldsymbol{\Theta} - \max(\boldsymbol{X}^T[m]\boldsymbol{\Theta}) \\ \sigma_M(\boldsymbol{x}, k) &= \frac{e^{\boldsymbol{x}[k]}}{\sum_{k'=1}^{K} e^{\boldsymbol{x}[k']}} \end{aligned} \tag{4}$$

where $\boldsymbol{x}$ is now the "normalized" matrix product of the inputs and the weights and k is the column index of the desired class. However, even with this more stable implementation, I was still encountering situations in which the softmax function would return 0, resulting in a log(0) calculation, which is undefined. To resolve this, I implemented the following function which combines the logarithm and the exponentiation into one calculation:

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{X}^T[m]\boldsymbol{\Theta} - \max(\boldsymbol{X}^T[m]\boldsymbol{\Theta}) \\ \log \sigma_M(\boldsymbol{x}, k) &= \log\left(\frac{e^{\boldsymbol{x}[k]}}{\sum_{k'=1}^{K} e^{\boldsymbol{x}[k']}}\right) = \left(\boldsymbol{x} - \log\left(\sum_{k'=1}^{K} e^{\boldsymbol{x}[k']}\right)\right)[k] \end{aligned} \tag{5}$$

In the code, Equation (4) can be seen implemented in the function `softmax(X, Theta, k)`, and is used when calculating the loss function gradient. Equation (5) can be seen implemented in the function `log_softmax(X,`

`Theta, k)`, and is used when calculating the value of the loss function.

In addition to computing the loss for each epoch, the loss function gradient must also be computed. From lecture, the gradient of the loss function with respect to each set of parameters $\boldsymbol{\Theta}_k$, given training data $\boldsymbol{D}$ as specified at the start of this section, is given as:

$$\frac{\partial L(\boldsymbol{D}:\boldsymbol{\Theta})}{\partial \boldsymbol{\Theta}_k} = -\sum_{m=1}^{M} \Big(\boldsymbol{Y}[m][k] - \sigma_M(\boldsymbol{X}^T\boldsymbol{\Theta}_k)\Big)\boldsymbol{X}[m] \tag{6}$$

Equation (6) above is implemented as written, noting that the softmax function used here is the numerically stable version described in Equation (4). The $\boldsymbol{\Theta}_k$ for each $k = 1, ..., K$ is computed, and then those column vectors are combined to create a loss gradient matrix, $\nabla\boldsymbol{\Theta}$. Then, the parameters $\boldsymbol{\Theta}$ are updated using:

$$\begin{gathered} \nabla\boldsymbol{\Theta} = \begin{bmatrix} \nabla\boldsymbol{\Theta}_1 & ... & \nabla\boldsymbol{\Theta}_K \end{bmatrix} \\ \boldsymbol{\Theta}^t = \boldsymbol{\Theta}^{t-1} - \eta\nabla\boldsymbol{\Theta} \end{gathered} \tag{7}$$

where η is the learning rate. For this project, a learning rate of 0.01 was used (how this value was selected is discussed in the last section).

Pseudocode

The following pseudocode outlines the algorithm implemented for training this model and saving all of the related data afterwards:

```
import training and testing data sets
reshape data into vector format
initialize Theta to random values in an N(0,1) distribution, initialize biases to 0

begin training loop
    for epoch = 1 to 25
        compute training loss
        compute training accuracy for each class
        compute testing loss
        compute testing accuracy for each class

        for k = 1 to 5
            compute loss gradient with respect to Theta_k
        end loop
        concatenate gradients to create gradient matrix

        update weights Theta
    end loop

save weights to file "multiclass_parameters.txt"
plot weights as images
plot training/test loss, training/test error
```

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 epoch is show in Figure 1.

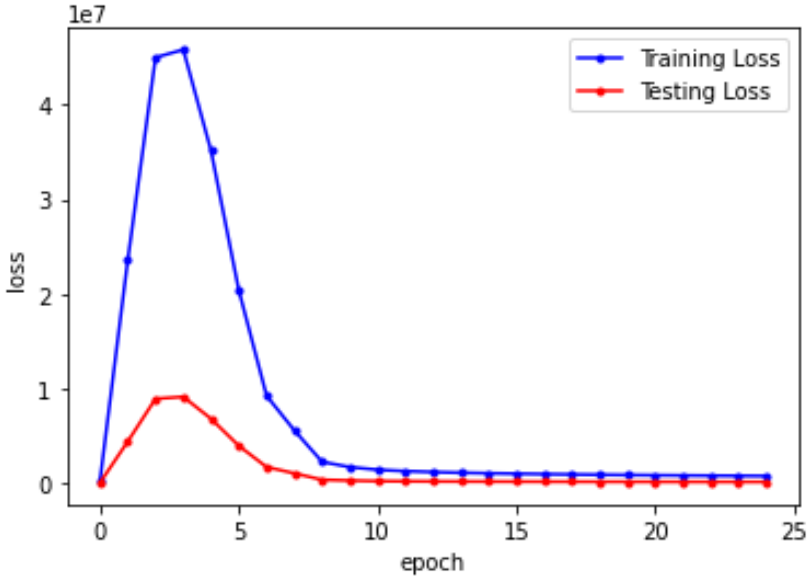


Figure 1: Model loss plotted as a function of epochs

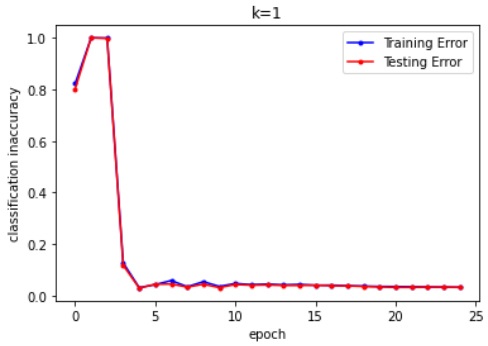
The above plot shows the loss of the model on the y-axis and the number of epochs elapsed on the x-axis. It shows that the training and testing loss both increased significantly at the start of training (as the model was just starting to adjust the initial randomized weights) and then decreased steadily and stabilized as more epochs completed. The fact that the testing loss curve follows the training loss curve is good, since it indicates that the model is not overfitting.

In addition to the loss values, the classification accuracy for each class was also evaluated after the completion of each epoch. 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 1 and the plots in Figure 2 are the ratio of incorrectly classified images for a specific class to the total number of images that appeared for that class.

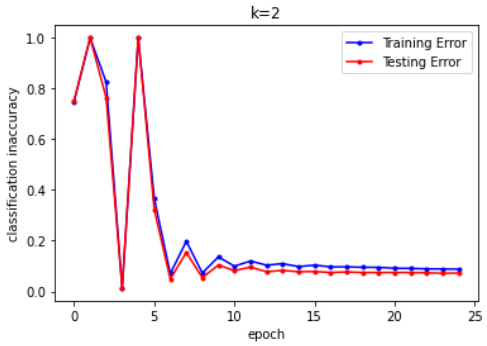
Class	Final Testing Inaccuracy
1	0.0320 (3.20%)
2	0.0727 (7.27%)
3	0.0796 (7.96%)
4	0.0336 (3.36%)
5	0.1060 (10.60%)
Average	0.0648 (6.48%)

Table 1: Final testing inaccuracy for each class and the overall average

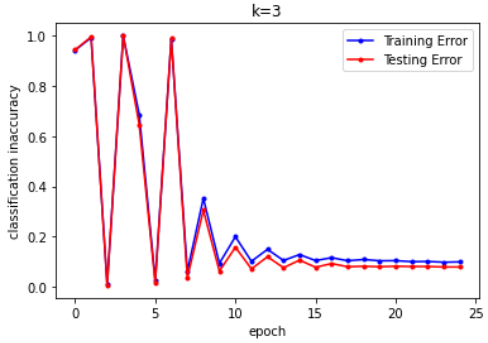
From Table 1, it is clear that this model is relatively accurate when making classification predictions. It seems that class 5 is the most problematic, with 10.6% inaccuracy, but this is likely due to the complexity of the digit itself, and how similar it is to the numerals 2 and 3 (both of which also had comparatively high inaccuracy rates). The change in these inaccuracy values for each class is plotted in Figure 2 below.



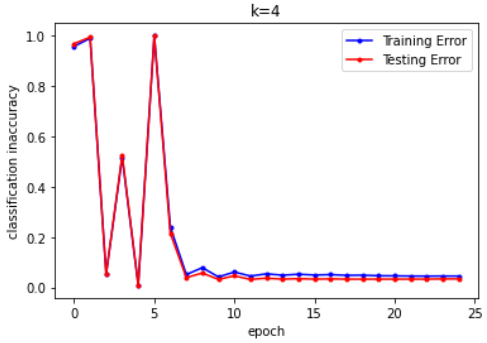
(a) Training and testing inaccuracy, k=1



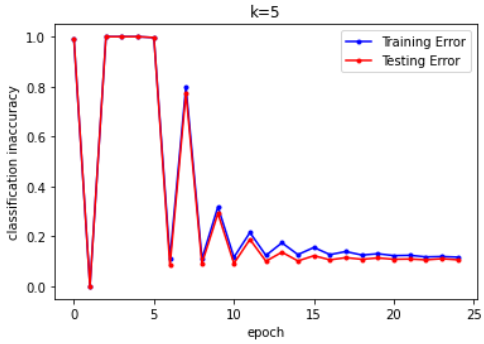
(b) Training and testing inaccuracy, k=2



(c) Training and testing inaccuracy, k=3



(d) Training and testing inaccuracy, k=4

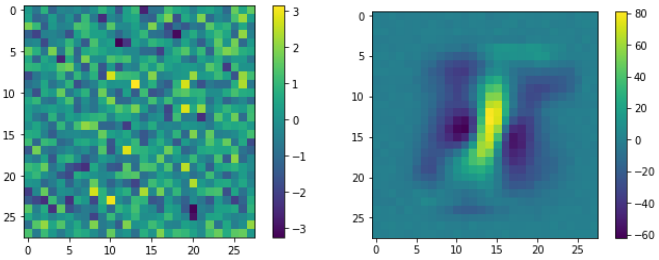


(e) Training and testing inaccuracy, k=5

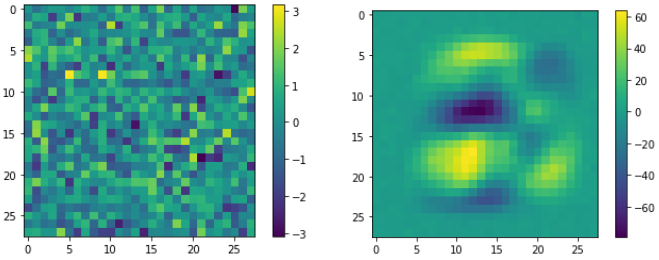
Figure 2: Training and testing accuracy for each class

The plots above show that for all classes except $k = 1$, the inaccuracy of the model started very high, zig-zagged from high to low for a few epochs, and then stabilized a low value and gradually decreased until the end of the training. For the $k = 1$ class, the inaccuracy dropped immediately and remained low, likely because the shape of a "1" numeral is very simple to recognize. The zig-zagging seen in these plots is likely due to the model initially overfitting, and this may be avoided with a much smaller learning rate (but that would require many more epochs to converge). The plots also show that the training inaccuracy and the testing inaccuracy follow each other very closely, which again is good and suggests that the model is not overfitting when it converges.

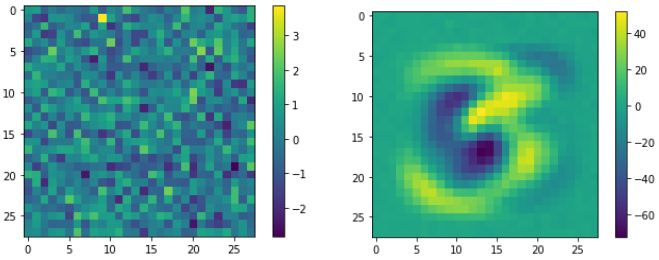
For reference, the weights for each class have been plotted as images before and after training completed. These images can be found in Figure 3 below. Note that the final weights resemble the digits that they are attempting to classify. This makes sense, since the image is showing what pixels of the image will be "worth" more and less to the classifier when analyzing the input.



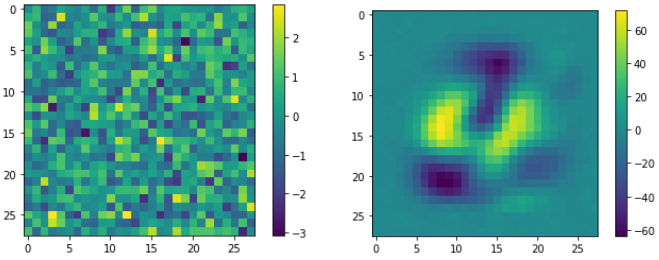
(a) Initial Weights, k=1 (b) Final Weights, k=1



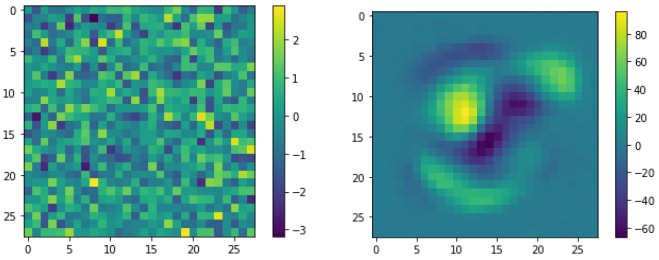
(c) Initial Weights, k=2 (d) Final Weights, k=2



(e) Initial Weights, k=3 (f) Final Weights, k=3



(g) Initial Weights, k=4 (h) Final Weights, k=4



(i) Initial Weights, k=5 (j) Final Weights, k=5

Figure 3: Weights for each class plotted as images

Discussion

As is evident from the last section, the model training process was smooth and the model performed as expected. This is in part due to the hyper parameters and settings that were used. In this section, I will detail each hyper parameter and setting and discuss how it was selected.

Learning Rate

For this project, a learning rate (η) of 0.01 was selected. When my program was completed, I ran it a few times with different learning rates to see what would work best. I tried to use 0.1, 0.01, 0.08, and 0.001 (each allowing the model to train for 25 epochs). It was clear that $\eta = 0.1$ was too large, as the model would overfit and oscillate for a long time instead of converging. Using $\eta = 0.08$ was also too large, as it would eventually converge but the inaccuracy rate was a bit larger than the inaccuracy rate of using $\eta = 0.1$. Lastly, using $\eta = 0.001$ also converged and had acceptable accuracy, but its path to convergence took a much longer time, and the resulting weights were much more noisy than the weights obtained using $\eta = 0.01$.

Number of Epochs

The number of epochs used in this project was 25. This number was selected after testing the model's performance using different epoch counts. I found that using 10 epochs allows the model to start to converge, but the inaccuracy rate remained relatively high. I also found that using 50 epochs caused the model to overfit, where the resulting inaccuracy rate was higher than the rate obtained when using only 25 epochs. I eventually decided to use 25 since this didn't allow the model to overfit but gave it plenty of time to converge (as seen in the model loss plot, where the value of the loss function bottoms out and slowly but steadily declines).

Initial Weight Values

The initial values for $\boldsymbol{W}$ were chosen to be random, non-zero elements (with exception of the biases, $\boldsymbol{W}_0$, which were all initialized to 0). This choice was largely due to the recommendation of this method in the lecture notes. The random elements were selected from a Gaussian distribution, i.e. $\boldsymbol{W}_k[i] \sim N(0, 1)$.