Softmax Regression Neural Network

Jack Kai Lim
Halıcıoğlu Data Science Institute
University of California, San Diego
jklim@ucsd.edu

Vivian Chen
Halıcıoğlu Data Science Institute
University of California, San Diego
vnchen@ucsd.edu

Abstract

In this project we implemented a neural network with a softmax regression for the output layer, from scratch using only numpy. The network was also trained using mini-batch SGD with momentum and L2 regularization abilities. In addition to that we also did a numerical approximation test to determine the correctness of our backpropagation implementation. We also did experiments with different activation functions, regularization methods and momentum values to explore the effects of each of them on the network. In the end the best test accuracy was $0.9759 \equiv 97.59\%$ and a test loss of 0.080695664408.

1 Dataset and Preprocessing

The dataset that was used for this project was **MNIST** dataset which can be found here. The dataset is a collection of handwritten digits ranging from 0 - 9, and have been size normalized and centered in a 28 x 28 pixel image. The dataset is split into 60,000 training images and 10,000 testing images.

1.1 Train, Test and Validation Split

In addition to the Train and Test split that was provided by the dataset, we also split the Training Data in a 80/20 split to create a Validation set. This was done to help us determine the best hyperparameters for our model.

This was done by taking the first getting the number of images in the training set, and getting an array of all the indices. Then we shuffled the indices and took the first 80% of the indices to be the training set, and the last 20% to be the validation set.

1.2 Normalization Procedure

The data was normilized by getting the mean by getting all the pixel values and computing the average pixel value. Then we subtracted the average pixel value from each pixel value and divided by the number of pixel which for a 28 x 28 image is 784, but to make the code work in general we used the shape of the image to get the number of pixels. The same procedure is done where we get the standard deviation of the pixel values across all the pixel values.

This was done by defining a function which normalizes a single images and then iterating through all the images from the input array and normalizing them.

1.2.1 Mean and Standard Deviation Example

```
Mean: 0.13714225924744897
Standard Deviation: 0.31112823799846606
```

This is the mean and standard deviation for the first image of the training set.

2 Softmax Regression

For the implementation of the Softmax Activation Function, we used the following formula:

$$y_k^n = \frac{e^{a_k^n}}{\sum_{i=k'}^{n} e^{a_i^n}} \tag{1}$$

Where y_k^n represents the output of the neuron after the softmax activation function is applied. a_k^n represents the input to the neuron. k represents the label of the classification and n represents the number of classes, which in turn also represents the dimension of the vectors. k' is defined to differentiate with the k in the numerator as we divide each class k by the sum of all the classes exponentiated. And finally a represents the input to the neuron.

2.1 Softmax Overflow Handling

Firstly the reason for the overflow where the denominator becomes 0, happens when the exponentiated sum becomes too large causing the denominator to become 0. To handle this we subtracted the maximum value of the input vector from each element of the input vector.

This works because the softmax function is invariant to constant offsets in the input vector. Which can be seen in the following equation:

$$\begin{aligned} y_k^n &= \frac{e^{\alpha}}{\sum_{i=k'}^{n} e^{\alpha}} = \frac{e^{a_k^n - c}}{\sum_{i=k'}^{n} e^{a_i^n - c}} \\ &= \frac{e^{a_k^n} e^{-c}}{e^{-c} \sum_{i=k'}^{n} e^{a_i^n}} \\ &= \frac{e^{a_k^n}}{\sum_{i=k'}^{n} e^{a_i^n}} \end{aligned} \tag{2}$$

Where c is the maximum value of the input vector.

Note: AI assistance was used to solve this problem. See Appendix

2.2 Clipping the Logits

For the Loss function which is the Cross Entropy Loss, via the following formula

$$E = -\sum_n \sum_{k=1}^{c} t_k^n \ln(y_k^n)$$

We came across an issue where the function would return NaNs. This is due to numerical instability where there is the possibilities for values like $\ln(0)$ and $\ln(-1)$ which are undefined, or when a logit becomes and extremely small number that is not 0, or an extremely large number which python cannot handle.

So inorder to handle this we clipped the logits to be between $[10^{-10}, 1 - 10^{-10}]$. This was done by using the following code:

```
clipped_logits = np.clip(logits, 1e-10, 1 - 1e-10)
```

Note: AI assistance was used to solve this problem. See Appendix

Results

The hyperparameters used for the training for the Neural Network with mini-batch SGD and Softmax Regression are as follows:

```
learning_rate = 0.01
batch_size = 128
```

```
epochs = 100
early_stop = True
early_stop_epochs = 3
```

```
Test Accuracy: 0.9214
```

The plots for the loss and accuracy can be seen in Figure 1 and Figure 2.

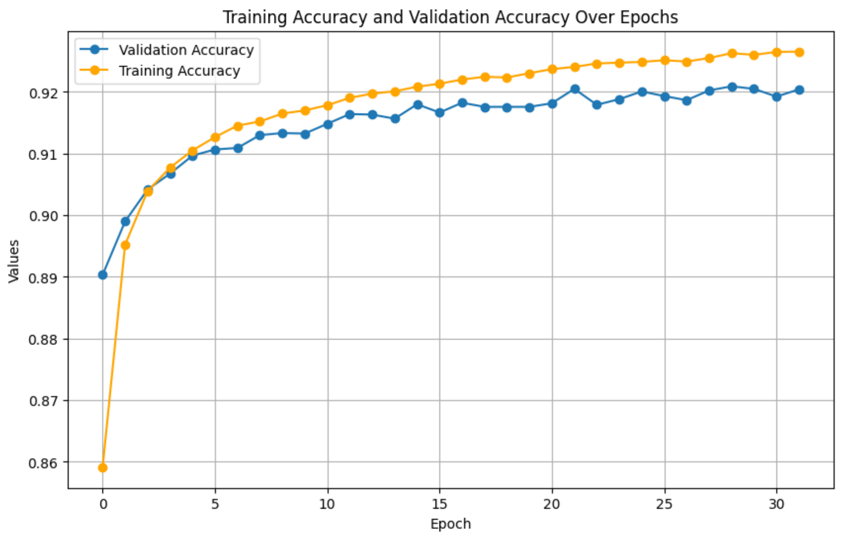

Figure 1: Train/Validation Accuracy for Softmax Regression

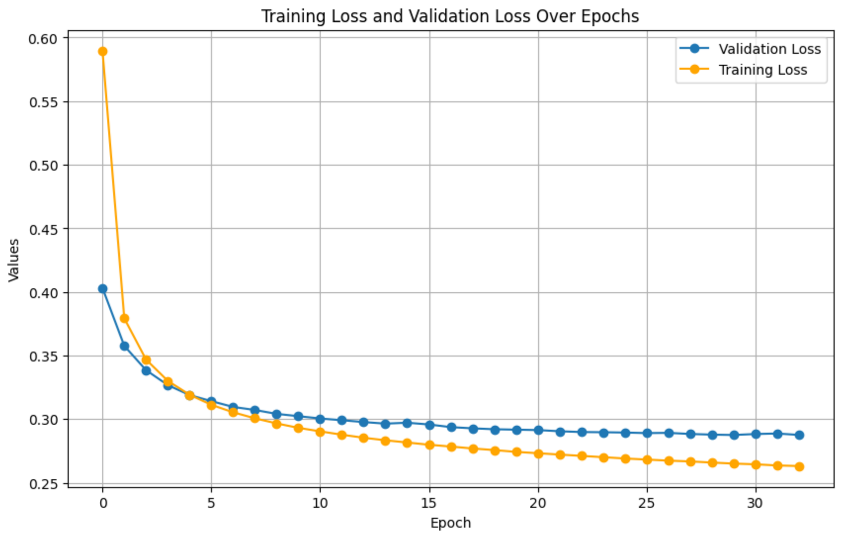

Figure 2: Train/Validation Loss for Softmax Regression

3 Numerical Approximation of the Gradient

To check whether our implementation of the backpropagation is correct, we will use numerical approximation to check the weights of the network. The numerical approximation is done by using the following formula:

$$\frac{\partial E^n}{\partial w}(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon} \tag{3}$$

Table 1: Numerical Approximation of the Gradient

Layer	Weights	Numerical Approximation	Backpropagation	Difference	Within ϵ^2?
Output	Bias	0.099881709278	0.099880510195	$1.19908287898e^{6}$	Yes
Output	Weights 1	0.0097590626103	0.0097590614918	$1.4609340039e^{-10}$	Yes
Output	Weights 2	0.004951752628	0.00495175248	$1.4609340039e^{-10}$	Yes
Hidden	Bias	0.0018200287225	0.0018200877310	$5.900851056e^{-8}$	Yes
Hidden	Weights 1	0.0009002833399	0.0009002904815	$7.1416564615e^{-9}$	Yes
Hidden	Weights 2	0.0009002833399	0.0009002904815	$7.1416564615e^{-9}$	Yes

After perfoming the numerical approximation with $\epsilon = 1e^{-2}$, these are the weights for a few selected weights in the network that we decided to check:

As we can see from Table 1 the we can see that all the backpropagation values are within ϵ^2 of the numerical approximation. This means that our implementation of the backpropagation is correct.

The hyperparameters for the network with mini-batch SGD are:

```
activation = 'tanh'
learning_rate = 1
batch_size = 128
epochs = 100
early_stop = True
early_stop_epochs = 5
L2_penalty = 0.0
momentum = False
momentum_gamma = 0.9
```

We also trained it on only 1 training sample.

4 Momentum Experiments

For the momentum experiments first we used a mini-batch SGD with the hyperparameters given in **config_6.yaml** which are as follows:

```
learning_rate = 0.01
batch_size = 128
epochs = 100
early_stop = True
early_stop_epochs = 3
regularization_type = None
L2_penalty = 0.0
L1_penalty = 0.0
momentum = True
momentum_gamma = 0.9
```

This yielded a test accuracy of 96.68 and the plots for the loss and accuracy, they can be seen in Figure 3 and Figure 4.

The training took all 100 epochs and never reached the early stopping condition. And looking at the graphs, both the training and validation accuracy and loss experienced similar trends, where they are constantly getting higher for accuracy and lower for loss, showing that the low learning rate and high momentum allowing the model to search for the best solution and converge to it.

Then we also played around with the momentum gamma and hyperparameters and we tried a combination where we had a small momentum gamma and a large learning rate. The hyperparameters used are as follows:

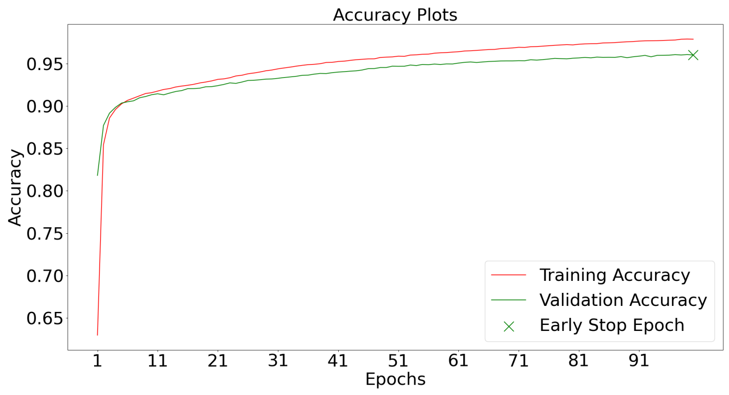

Figure 3: Train/Validation Accuracy for Momentum

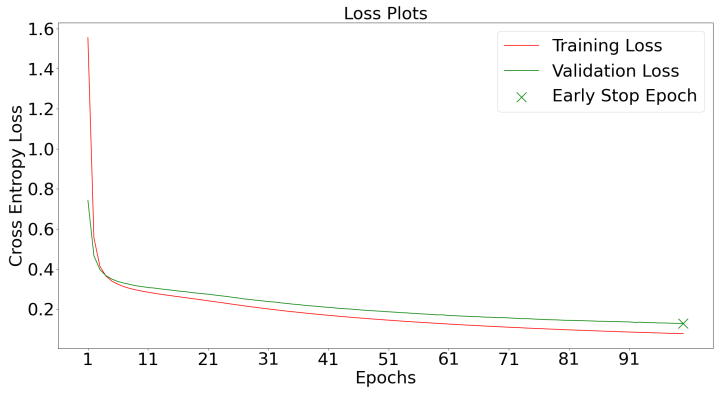

Figure 4: Train/Validation Loss for Momentum

```
learning_rate = 0.01
batch_size = 128
epochs = 100
early_stop = True
early_stop_epochs = 3
regularization_type = None
L2_penalty = 0.0
L1_penalty = 0.0
momentum = True
momentum_gamma = 0.01
```

This yielded a test accuracy of 91.87 instead of the 96.68 from the previous test. This is due to the low momentum gamma that was set for the second test, this is because, with lower momentum gamma, it makes the network more sensitive and will usually prevent the chance of overshooting the minimum. However, it will also lead to slower convergence which is why we see a lower test accuracy as compared to the first test.

Overall, the second test gave us a more stable and reliable model, as compared to the first test, which is why we would prefer the second test over the first test if our only goal is to achieve a higher test accuracy and the best possible model. But the tradeoff will be it's slower convergence rate.

The training plots for this can be seen in Figure 5 and Figure 6.

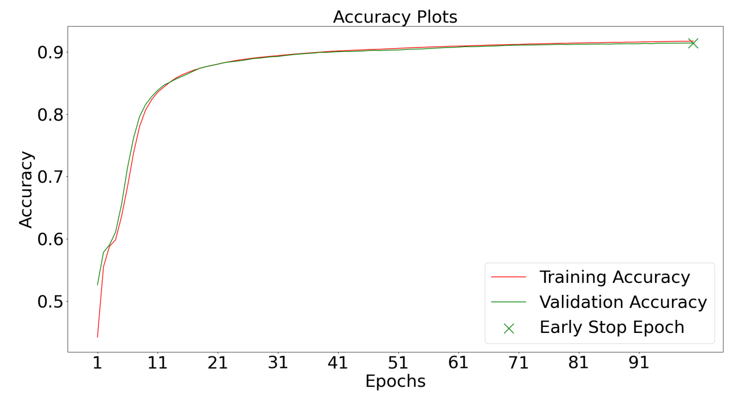

Figure 5: Train/Validation Accuracy for Momentum

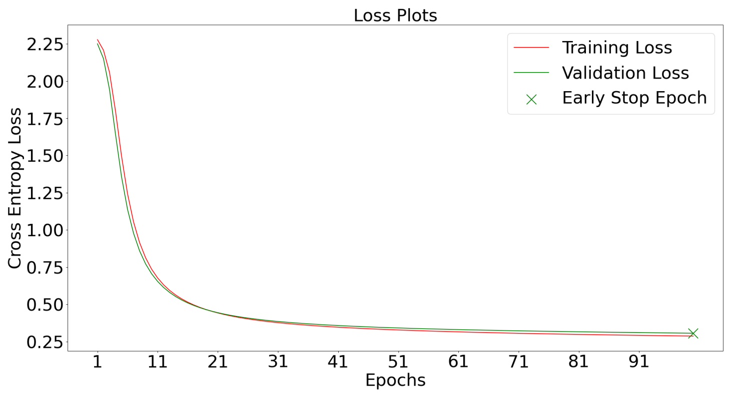

Figure 6: Train/Validation Loss for Momentum

5 Regularization Experiments

In this section we did experiments with different regularization methods which are L2 regularization and L1 regularization.

5.1 L2 Regularization

For L2 regularization we SGD with mini-batches and used the following hyperparameters:

```
learning_rate = 0.01
batch_size = 128
epochs = 110
early_stop = True
early_stop_epochs = 3
regularization_type = 'L2'
L2_penalty = 0.01
L1_penalty = 0.01
momentum = True
momentum_gamma = 0.9
```

This yielded a test accuracy of 0.9699 and a test loss of 0.10277824069050659 the plots for the the loss and accuracy can be seen in Figure 7 and Figure 8.

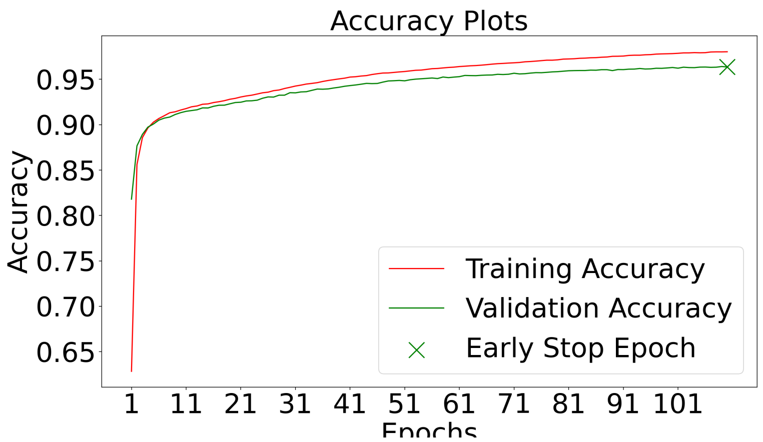

Figure 7: Train/Validation Accuracy for L2 Regularization

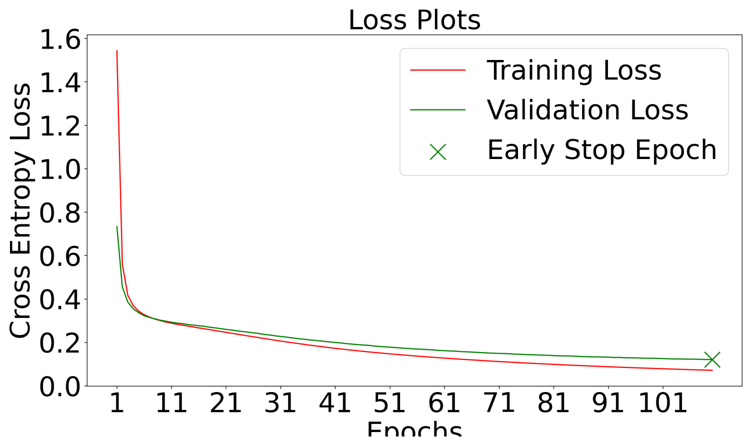

Figure 8: Train/Validation Loss for L2 Regularization

5.2 L1 Regularization

For L1 regularization we used the same hyperparameters as L2 regularization except we changed the regularization type to L1 and increased epochs from 110 to 160 to see if how much more decay L1 adds versus L2. This yielded a test accuracy of 0.9737 and a test loss of 0.08931316605518423 the plots for the loss and accuracy can be seen in Figure 9 and Figure 10.

5.3 L2 with smaller λ

For this experiment we used the same hyperparameters as the L2 regularization experiment except we changed the L2 penalty from 0.01 to 0.0001. This yielded a test accuracy of 0.9713 and a test loss of 0.10168974708457909, the plots for this can be seen in Figure 11 and Figure 12.

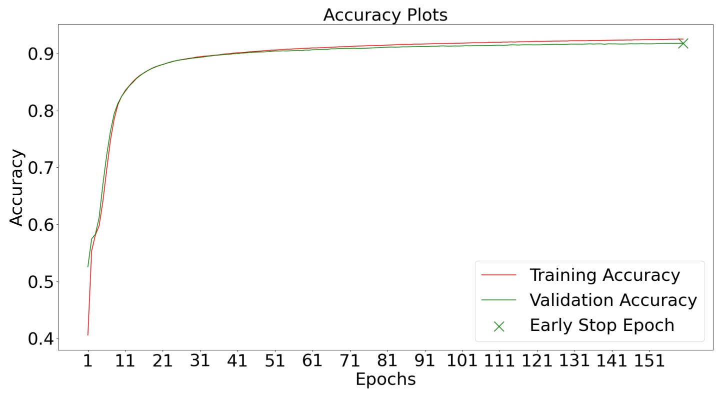

Figure 9: Train/Validation Accuracy for L1 Regularization

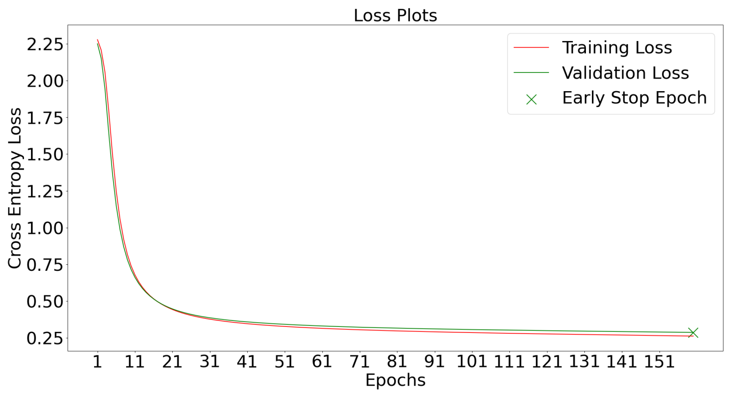

Figure 10: Train/Validation Loss for L1 Regularization

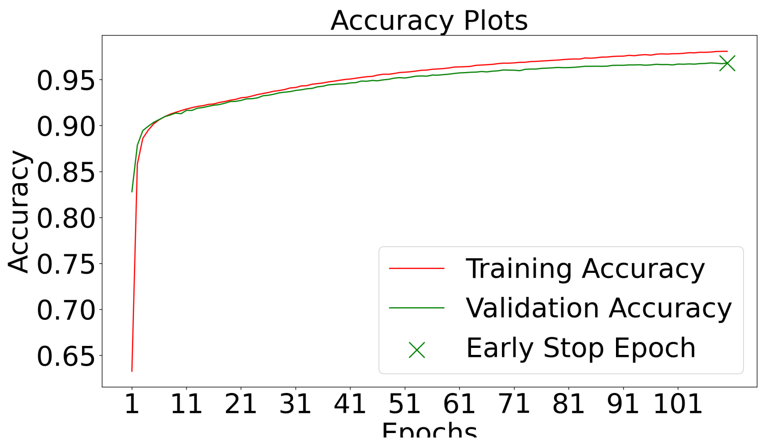

Figure 11: Train/Validation Accuracy for L2 Regularization with smaller λ

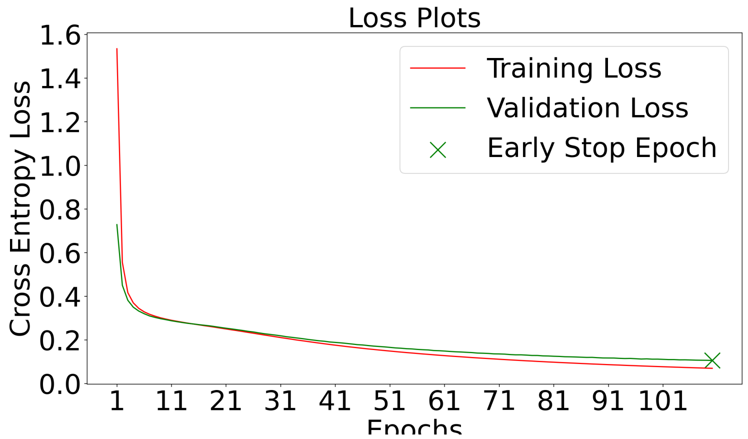

Figure 12: Train/Validation Loss for L2 Regularization with smaller λ

5.4 Observations

5.4.1 L2 vs L1 regularization

As can be seen on the both graphs, both the L2 and L1 accuracy graphs are very similar. Both lines on the graph are shaped like a negative exponential function with a positive intercept. The L1 regularization does have a slightly higher test accuracy than L2, but the difference is not significant since the accuracy difference is only around .004. Similarly, the loss graphs for both regularizations are similar since both are shaped as a e^{-x} graph. Also, the loss of L2 is higher than the loss of L1 by 0.01. One major difference between L1 and L2 regularization is that the L1 regularization required more epochs than L2 regularization to reach the early stopping condition.

5.4.2 Smaller λ vs Larger λ

Both the graphs for accuracy and loss for the smaller lambda vs the larger lambda for L2 regularization are very similar. The smaller lambda graph has a higher test accuracy than the larger lambda graph by 0.0013. Also, the loss of the smaller lambda is lower than the loss of the larger lambda by 0.001

6 Activation Experiments

In this section we did experiments with different activation functions using mini-batch SGD and the following hyperparameters for all the experiments:

```
learning_rate = 0.01
batch_size = 128
epochs = 100
early_stop = True
early_stop_epochs = 3
regularization_type = 'L2'
L2_penalty = 0.001
L1_penalty = 0.01
momentum = True
momentum_gamma = 0.9
```

With the layer sizes being $[784, 128, 10]$.

6.1 Sigmoid Activation

using the Sigmoid Activation function we got a test accuracy of 0.9336, and a test loss of 0.22483864970441. The training plots can be seen in Figure 13 and Figure 14.

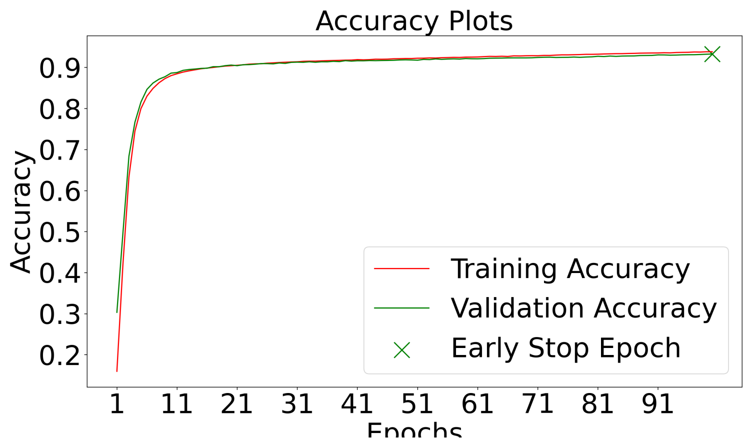

Figure 13: Train/Validation Accuracy for Sigmoid

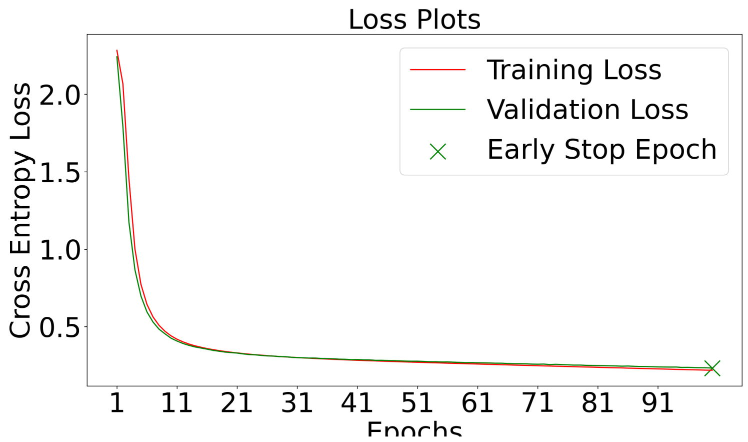

Figure 14: Train/Validation Loss for Sigmoid

6.2 ReLU Activation

For the ReLU Activation function we got a test accuracy of 0.9759 and a test loss of 0.080695664408. The training plots can be seen in Figure 15 and Figure 16.

6.3 Observations

From what could be observed, the sigmoid activation's test accuracy is lower than the ReLU activation's test accuracy, and the sigmoid activation's loss is higher than the ReLU activation's loss. Therefore, it could be implied that in this particular case, the using ReLU as an activation method is more effective than using sigmoid. Another aspect that could be observed is the fact that the both loss graphs are shaped as a negative exponential function. ReLU's losses are less than sigmoid's losses throughout each epoch. In terms of the accuracy graphs, all the lines are shaped as $-e^{-x}$ with a positive intercept. Similar to the loss graphs, ReLU's accuracies are more than sigmoid's accuracies throughout each epoch especially since the ReLU's graphs have a steeper increase.

7 Best Model

After some testing the model with the best test accuracy was the model with the following hyperparameters:

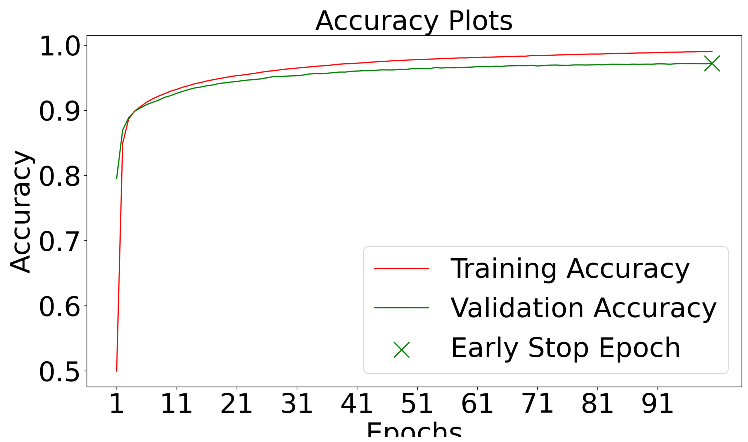

Figure 15: Train/Validation Accuracy for ReLU

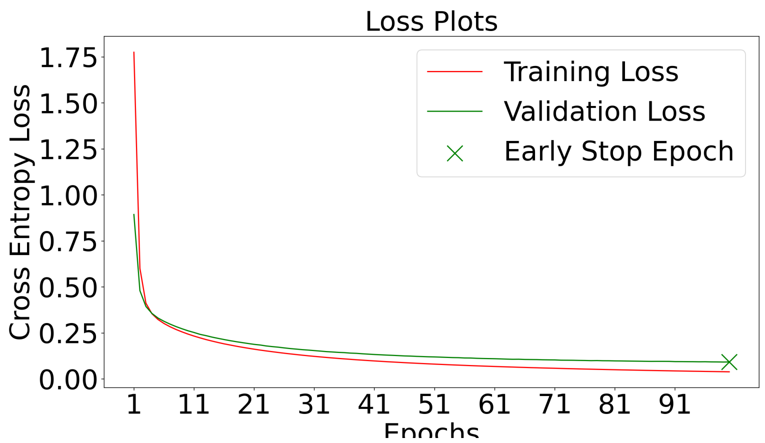

Figure 16: Train/Validation Loss for ReLU

```
learning_rate = 0.01
batch_size = 128
epochs = 100
early_stop = True
early_stop_epochs = 3
regularization_type = 'L2'
L2_penalty = 0.001
L1_penalty = 0.01
momentum = True
momentum_gamma = 0.9
```

With ReLU as the activation function, with a test accuracy of $0.9759 \equiv 97.59\%$ and a test loss of 0.080695664408.

8 Team Contributions

Both members contributed equally to the project. We worked together on the most if not all the code, only have offline work be mostly writing the report as Jack is fluent with LaTeXand we did the graphs offline mostly. Another aspect of offline work was debugging as we had many bugs and issues with questions 2 and 3 so some offline work from both members was done to debug and fix the issues.

Otherwise, all the work was done together when we met up in Geisel and worked on the project together.

A Appendix

A.1 AI Usage

Here are all the times AI assistance was used in this project (with the prompts)

A.1.1 Softmax Regression (Overflow Condition) - ChatGPT

Prompt: What condition in softmax regression would cause the denominator to become zero and how do we deal with it?

A.1.2 Clipping the Logits - ChatGPT

Prompt: We are trying to calculate the categorical cross entropy loss for a softmax regression, and we have the Loss function as the following

-np.sum(targets * np.log(logits))

howeverthis is return nans, why and what is causing this