Vision for CIFAR with BackProp

Authors: Hieu Luu, Michael Ruddy, Joseph Samuel

Abstract

In this project, we created implemented deep neural networks in order to classify images in the CIFAR-10 data set. In this project our main goal was to test how different network parameters(activation function, regularization, momentum) and architecture (number of layers and units) affected the accuracy when classifying images in the dataset. For the base model that made use of momentum, we got an accuracy of 41.91%. For the model that experimented with regularization, we got an accuracy of [41.55%]. For the model that experimented with different activations, we got an accuracy of 36.36% when using sigmoid and accuracy of 47.22% with ReLU. For the model that experimented with network topology, we got an accuracy of 40.24% when halving the number of hidden units (64 hidden units), an accuracy of 42.75% when doubling the number of hidden units (256 hidden units), and an accuracy of 38.85% when increasing the number of hidden layers to 2 (238 hidden units each).

1 Data Loading

We used the CIFAR-10 dataset.

1.1 Description

The CIFAR-10 data set has 10 different classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). It consists of 60,000 32x32 colour images (three channels for red, green, blue) with each class having 6,000 images. After processing, the images are turned into flattened arrays, with each image going from 32 x 32 x 3 to 1 x 3072.

1.2 Splitting

To generate validation data, we first shuffled the data and then split the training set into 80-20, with the 20% being the validation set. This was used to get an idea of how our model would perform on an unseen dataset during training. It allowed us to make our implementation was working and helped us tune the learning rate.

1.3 Normalization Procedure

Within each image, we normalized the pixel values on a per-channel basis. This meant that for each image, within each 32 x 32 channel (red, green, blue), we would subtract the mean value in that channel from each pixel and divide by the standard deviation within that channel. For one example:

$$\mu_{red} = 145.492 \qquad \mu_{green} = 86.305 \qquad \mu_{blue} = 69.394$$
$$\sigma_{red} = 59.988 \qquad \sigma_{green} = 60.285 \qquad \sigma_{blue} = 54.407$$

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

2 Numerical Approximation of Gradients

We used

$$\frac{d}{dw}E^n(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon}$$

to approximate the gradient of the error with respect to one weight. This allowed us to check the accuracy of our gradient calculated with back propagation. We did this for multiple weights within the network with $\epsilon = .01$. The differences are all within $O(\epsilon^2)$, indicating back propagation worked properly.

Type of Weight	Numerical Approx. Gradient	Backprop. Gradient	Absolute Difference
Output Bias	0.292	0.292	3.93e-8
Hidden Bias	0.273	0.273	3.41e-8
Hidden to Output	-0.0154	-0.0154	1.15e-8
Hidden to Output	-0.190	-0.190	3.96e-8
Input to Hidden	-0.0108	-0.0108	1.00e-8
Input to Hidden	-0.187	-0.187	4.27e-8

3

Training Procedure We used mini batch stochastic gradient descent with a momentum term weighted by $\lambda = 0.9$ in our update rule. Early stop patience was set to five epochs where training would halt if the loss function is not further minimized for five straight epochs. We were able to obtain a test accuracy of 41.91% using the following hyperparameters. Learning rate of 0.000004, 128 unit hidden layer, tanh activation function, regularization constant of 0, a batch size of 128, 100 training epochs, early stop with a patience of 5 epochs, momentum $\lambda = 0.9$, and random initial weights.

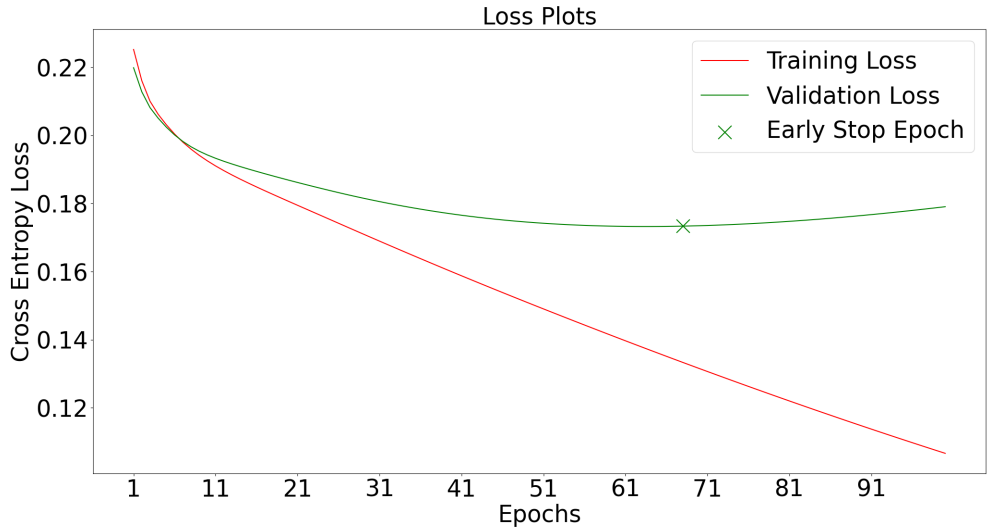

Figure 1: Loss values for model with momentum $\gamma = 0.9$

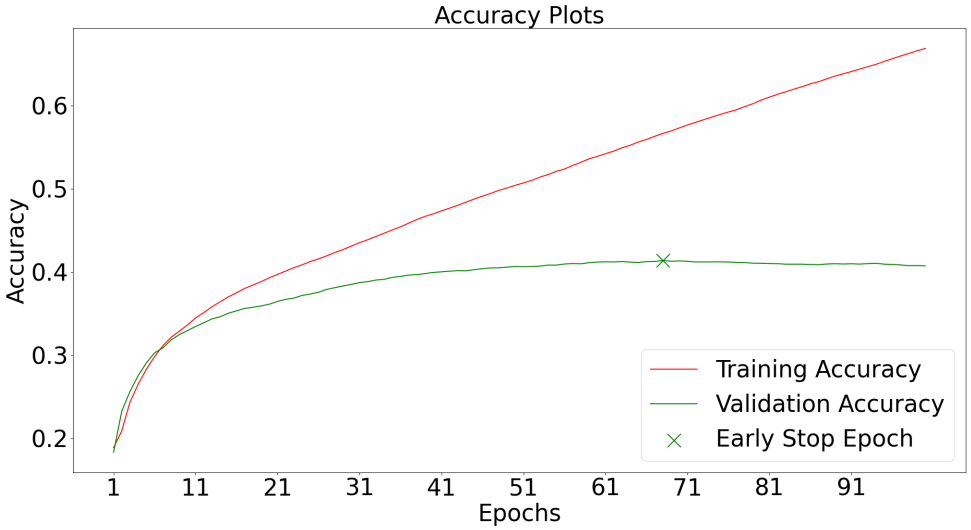

Figure 2: Accuracy values for model with momentum $\gamma = 0.9$

Observation and Inference: One observation we found was that the loss function was minimized at around 70 epochs into training at which point over fitting began. This allowed us to see how essential a good early stop function is to saving time and ensuring the best model when training. Another observation was that we had to adjust our learning rate to be lower than we expected. Upon further thought we inferred this was because we implemented our momentum as: $velocity = \gamma * velocity + learningrate * dw$ which gives the recurrence relation $R_t = \gamma * R_{t-1} + d$ that once solved gives a total sum of the coefficients of $1/1 - \gamma$ which in our case of $\gamma = 0.9$ makes our momentum term 9 times larger than the dw term, hence why we needed to decrease our learning rate accordingly.

4

Training Procedure We used the same model from the previous section with a few key changes. Instead of implementing early stop, we trained for a full 75 epochs which is what we found to be around 10% more than our early stop epoch in the previous section. We were able to obtain our highest test accuracy of 42.4% doing L1 regularization and our highest for L2 was 41.9%. We found both types of regularization did better with a larger λ while the L2 regularization did slightly better with a smaller λ. We found it odd that the L2 regularization had such a small impact on the test accuracy. We think this might be because our network has a large amount of feature and L2 doesn't minimize the impact of the less important features like L1 does.

L2 Test Accuracy: 41.9% $\lambda = 0.01$ 41.71% with $\lambda = 0.0001$

L1 Test Accuracy: 42.4% $\lambda = 0.01$ 41.53% with $\lambda = 0.0001$

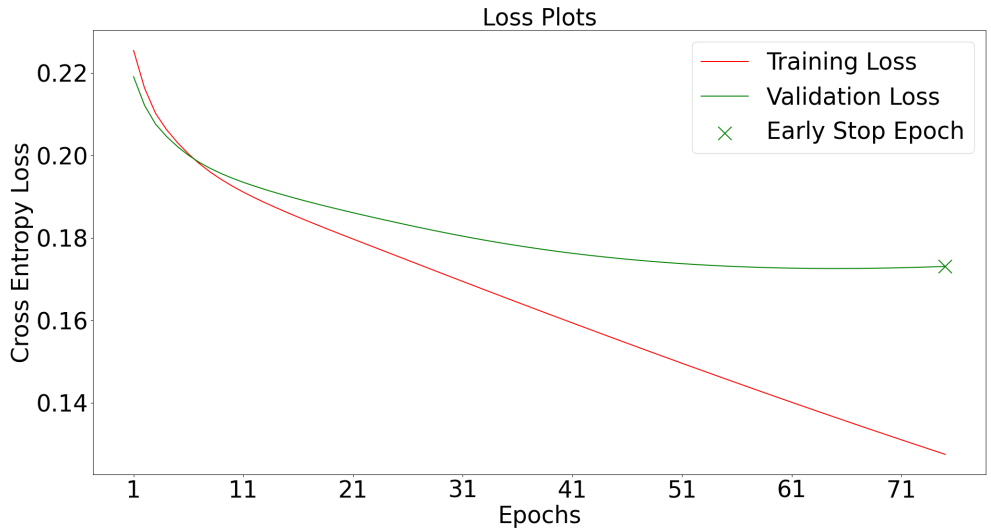

Figure 3: Loss values for model with regularization $\lambda = 0.01$

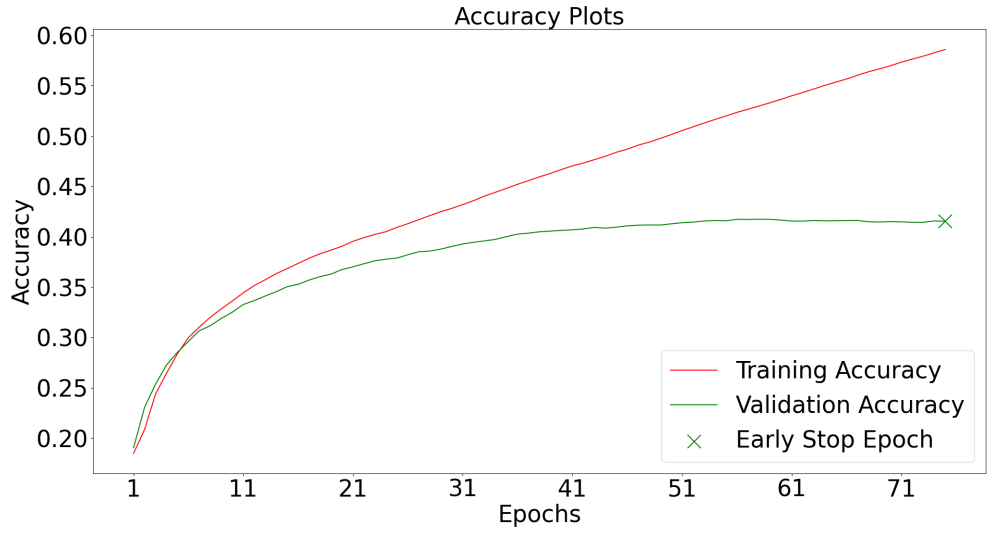

Figure 4: Accuracy values for model with regularization $\lambda = 0.01$

5 Different Activations

5.1 Sigmoid

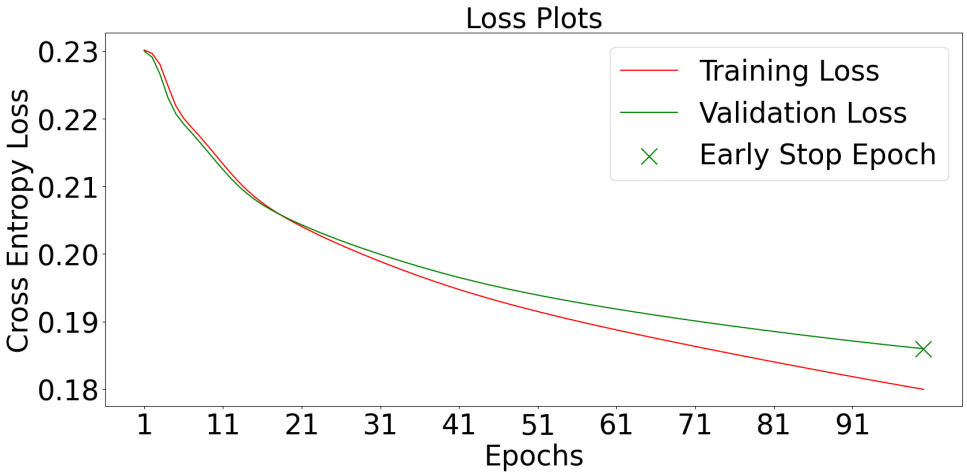

Figure 5: Loss values for model with sigmoid activation

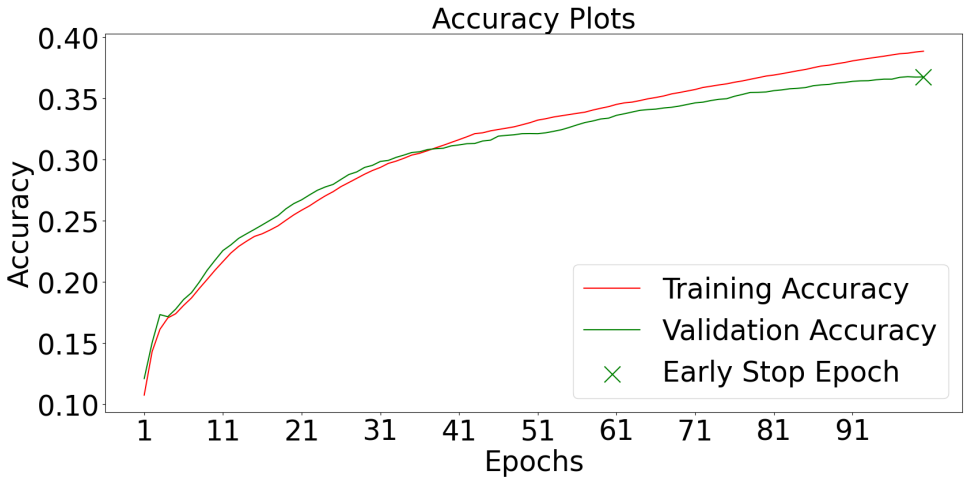

Figure 6: Accuracy for model with sigmoid activation

5.2 ReLU

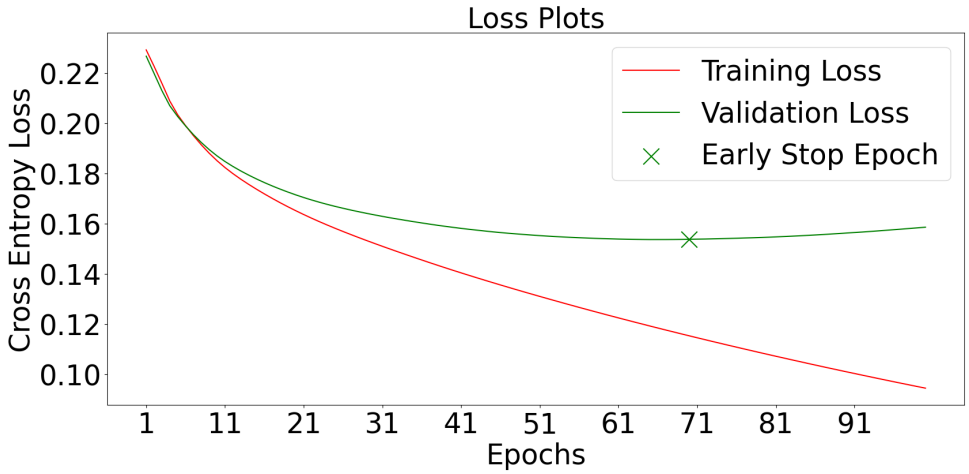

Figure 7: Loss values for model with ReLU activation

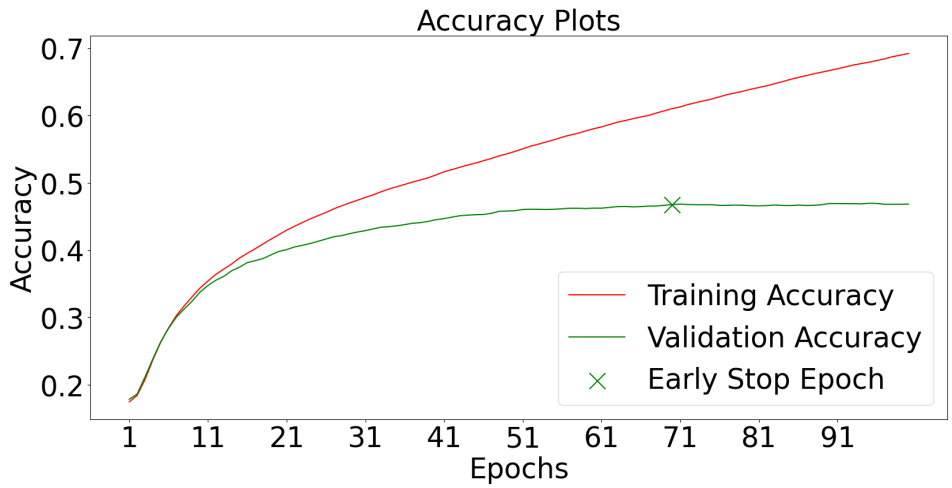

Figure 8: Accuracy for model with ReLU activation

5.3 Performance

Sigmoid final test accuracy: .3636
ReLU final test accuracy: .4722

Sigmoid performed much worse overall. This is due most likely due to it always producing a positive output and a vanishing gradient. With all positive outputs, the weight changes in the network are either all positive or all negative. This makes the network converge slower because the movement of the weights are restricted. Due to the slope of the sigmoid being close to 0 at the end ranges, it lessens the weight change significantly, further slowing down learning.
ReLU performed much better. It avoids the quickly vanishing gradient of the sigmoid and weight channges aren't as restricted either.

6 2f. Network Topology

6.1 i. Hidden units

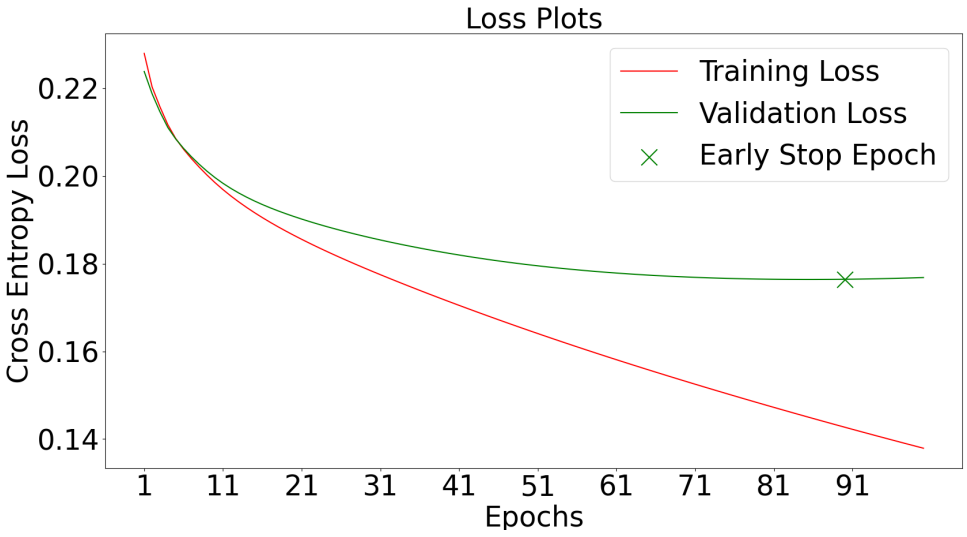

Figure 9: Loss values for model with 64 hidden units

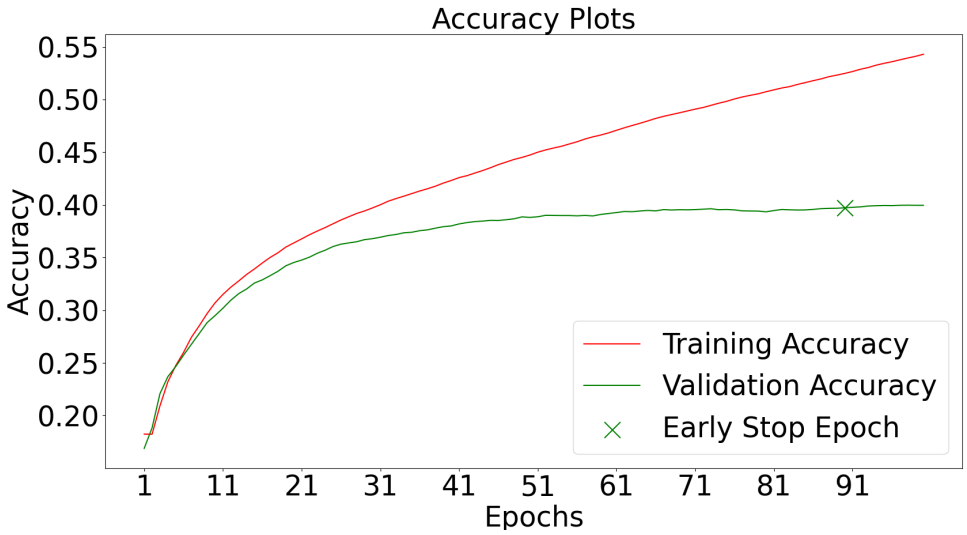

Figure 10: Accuracy for model with 64 hidden units

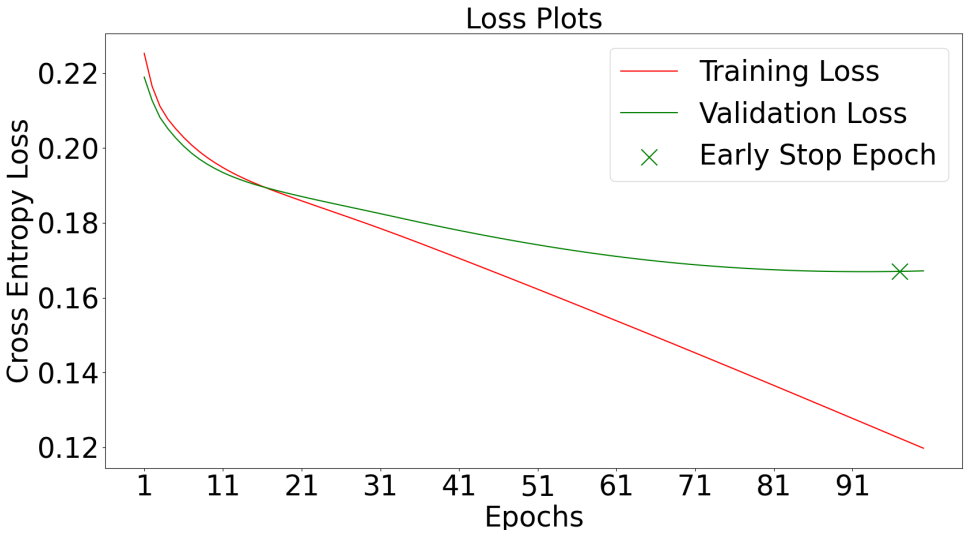

Figure 11: Accuracy for model with 256 hidden units

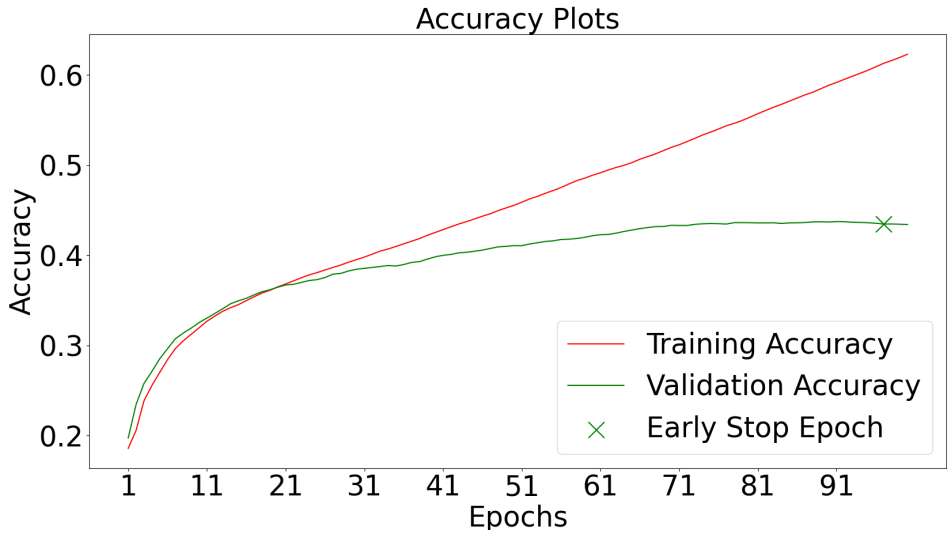

Figure 12: Accuracy for model with 256 hidden units

For the model where we cut the number of the units in the hidden layer in half from 128 units to 64 units. By cutting the hidden units in half to 64 the model was able to get a test accuracy of 40.24%. This is slightly worse than our base model which had 128 hidden units and achieved a text accuracy of 41.91%. The model with half the hidden wights also seemed to train faster but also converged faster when compared to the base model. For the model where we doubled the number of hidden units from 128 to 256 we were able to get a test accuracy of 42.75%. This is slightly better than our base model which had 128 hidden units. We also saw that the model with double the hidden units took longer to train and also took longer to converge. In the end a model with fewer hidden units is able to train faster since it has less operations(less weights to update in model), but it also has less accuracy since there are less hidden units which means it can not learn as many features.

6.2 ii. Hidden layers

Figure 13: Loss values for model with 2 hidden layers

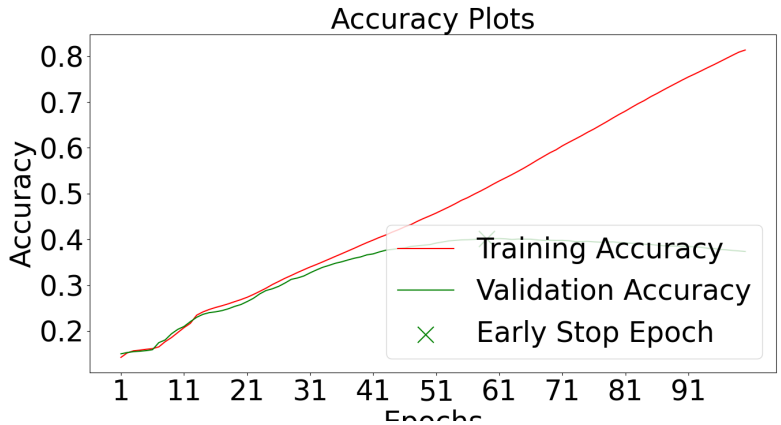

Figure 14: Accuracy for model with 2 hidden layers

For the 2 hidden layer network we used a network architecture of [3072, 238, 238, 10] for a total parameter count around 790,398. The best model in our previous experiment was a [3072, 256, 10] network which had a total parameter count of around 792,331. In our 2 hidden layer network we got a test accuracy of 38.85%. The 2 hidden layer network seems to train much slower than the best model in our previous experiment. It also has a sizeable drop in accuracy. An inference on why this may be the case is that the 2 hidden layer network my be over fitting the data since it can extract more features than the best model in our previous experiment (single hidden layer network). This over fitting leads to a drop in test accuracy since the 2 hidden layer now poorly generalizes.

7 Team Contributions

Joseph Samuel: For this project I coded up the append_bias, createTrainValSplit, all the activation functions and their gradients except the output function. I also assisted with the coding of the backprop and forward for both the layer and network classes. Lastly I did the experiments for network topology (2f) and wrote the abstract and 2f parts of the report

Michael Ruddy: I coded train.py, the normalization method, the load data method, helped code trainValSplit and tune learning rate for model in 2c. I also helped implement regularization for 2d. I did 2c and 2d on the report.

Hieu Luu: I helped code forward and backward in layer and neuralnet, normalization and train.py. I also did gradient.py for Numerical Approximation of Gradients. I helped implement momentum and tuned the learning rate to get the final model in 2c. I also did experiments with activations. Lastly, I wrote the data loading, 2b, and 2e part of the report.