Backpropagation for Multi-Layer Neural Networks With Softmax Outputs

Ramtin Kazemi
Student
UCSD
rkazemi@ucsd.edu

Rajdeep Nag
Student
UCSD
rnag@ucsd.edu

Henry Austin
Student
UCSD
hdaustin@ucsd.edu

Abstract

In this Assignment we used backpropagation to create a multi-layer neural network with one hidden layer with softmax outputs to categorize a given set of CIFAR-10 test data into the right type of image classes. We used a large set of training data from the CIFAR-10 dataset to train our model and tested this against the smaller testing set to report the loss, more specifically, the cross entropy loss and target accuracy for the test set. We fine tuned our models with adding early stopping, momentum, as well as L2 and L1 regularization to increase the accuracy and reduce the loss while testing on our model. We further experimented with different activation functions for the hidden layer units, and the topology of the Neural Network (i.e. number of hidden layers, number of hidden units). Our accuracies are listed below.

Accuracy for test set on Neural Network (minibatch stochastic gradient descent with backpropagation) with Momentum: 40.97%

Best accuracy for test set on Neural Network with Regularization: 45.5% (L1)

Best accuracy for test set on Neural Network with Activation Function experimentation: 41.36% (Sigmoid)

Best accuracy for test set on Neural Network with Network Topology experimentation: 43.89% (Single hidden layer with 256 units)

1 Introduction

The dataset given to use was 40,000 images for training, 10,000 for validation and 10,000 for testing. The dimensions and more information about the dataset is further explained in section 3. Dataset.

Our multi-layer neural network had one layer of hidden units for which the activation function was default ‘tanh’ but we have also performed experiments using ‘ReLU’ and ‘sigmoid’. The output layer’s activation function was softmax as we wanted to categorize the outputs into 10 classes corresponding to the image classes in the CIFAR-10 dataset.

Our code structure divided the neural network into 3 main classes, the unit class, the layer class and the entire neural network as a class of its own. The neural network iterated through each layer wherein both forwarding and backpropagation was done wherein we performed activation on each unit during forwarding and found the gradient (slope) of each unit during backwarding to find the change in weights for each layer.

Before training and testing our model we checked if the gradients with respect to weights for each layer calculated during backpropagation was the same as its numerical approximation for a single weight. Then we tested several models with the testing set with different hyperparameters,

activation functions, momentum and regularization.

2 Related Work

The resources our group used throughout the duration of this assignment were lecture and discussion slides, supplemented with Piazza forum posts and the NumPy and MatPlotLib public documentation. We also used the OH's with several TA's which were very helpful. The specific PowerPoints used from the course included: Lecture 3, Lecture 4A, Lecture 4B. These lecture resources helped us understand how to implement backpropagation as well as SGD. Lectures 4A and 4B were crucial in us being able to complete momentum and regularization.

3 Dataset Classification

The dataset that was provided to us is known as CIFAR-10 which was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. We performed the multi-layer algorithm with backpropagation on this dataset where we used softmax to classify the testing data into their corresponding classes and check for the accuracy of this. Our training set was of the size of 50,000 images of size 28 by 28 pixels in three channels, namely, red, green and blue. However each of these images was stored in linear fashion with 3072 dimensions representing each pixel. We split this training dataset into 80% and 20% for training and validation. The outcome of this was 40,000 images for training, 10,000 images for validation and an additional 10,000 images for testing. As we used softmax outputs we also one-hot encoded our targets so we can check if it matches with the highest probability class for each image. We normalized the data using z-score normalization for each image along each channel such that each image would have a mean of 0 and unit variance along each of the three channels. Each pixel value after normalization was roughly within the bounds of [-1, 1].

4 Numerical Approximation of Gradients

We calculated the numerical approximation of the loss using the equation:

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

where E(w) is the loss for a particular weight and it's derivative would give us the gradient with respect to weights. Here the value of epsilon used was 0.01 or 10^{-2}.

Type of weight	Actual Gradient	Numeric Approximation	DIfference
Hidden Bias	0.0000123	0.0003229	0.000311
Output Bias	0.000295	0.000189	0.000104
Input to Hidden Weight 1	0.000134	0.0000387	0.0000958
Input to Hidden Weight 2	0.000056467	0.001089	0.00103
Hidden to Output Weight 1	0.000198	0.00126	0.00106

Hidden to Output Weight 2	0.000866	0.000554	0.000312

Here, you can see that the differences between all the approximations and actual gradients all lie between $O(10^{-2})$ *and* $O(10^{-4})$.

3 Neural Network with Momentum

Best Model:

Momentum 100 epochs 0.01 Learning Rate Batch Size = 128

Test Accuracy: 40.97 Test Loss: 1.7545736208681992

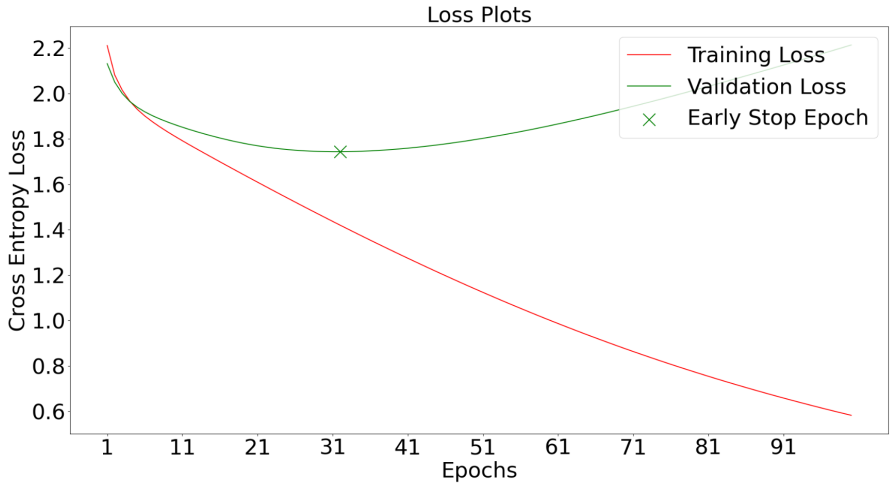
Other tests:

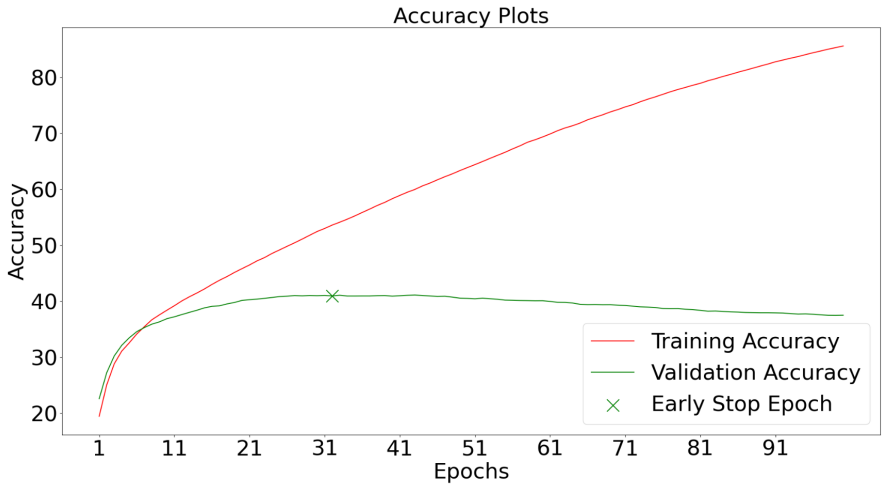
Momentum 100 epochs 0.01 learning rate Batch Size = 64

Test Accuracy: 40.61 Test Loss: 1.7552282564046617

Best Model Plots:

Training vs Validation Plots with momentum gamma = 0.9





Training procedure: Our training procedure involved training the data using SGD(stochastic gradient descent) in minibatches over a certain amount of epochs. More specifically, for every epoch: we shuffle the data and generate the minibatches; then for each minibatch: we call forward and backward on our Neural Network (where forward computes the forward activation pass, and backward computes the error backpropagation, changing the weights). We also store the loss and accuracy of every epoch for plotting. We have also implemented early stopping, where if the validation loss increases for "patience" epochs (where patience is defined as a hyperparameter integer, we set it to 5), then the training procedure stores a copy of the model and its parameters at the epoch before the increase that triggered the 'early stop.' We also implemented momentum which helped to increase how quickly our model learned, increasing our accuracy and reducing loss. Momentum moves the Network in the direction of most consistency in the gradients, rather than jumping towards the true gradient every time; this is why our loss became smoother and reduced greatly.

Discussion of choice of hyperparameters:

In this portion of the assignment, we used 3072 inputs, 128 hidden units (in a single layer), and 10 output units. We used a batch size of 128, a learning rate of 0.01, and we trained the data for 100 epochs. Using a lower learning rate of 0.001 we saw that the validation curves were closer to the training curves but this actually lowered our testing accuracy to around 38%.

Test accuracy: For this test, we were supposed to be able to achieve an accuracy of about 44% on the given training and testing data. We achieved a testing accuracy of roughly 41% and we believe this is proof that our Neural Network is working without bugs!

Observations and inference from the experiments:

During testing, we had an issue where our loss plot would explode. After combing through our code and consulting a TA, we discovered that we needed to divide our gradient by the batch size. This helps to avoid your loss values from growing too big due to the gradient growing too large!

4 Regularization Experiments

Best Model:

L1 Regularization: with $\lambda = 10^{-2}$ Test Accuracy: 45.5 Test Loss: 1.5793496173086166

Other tests:

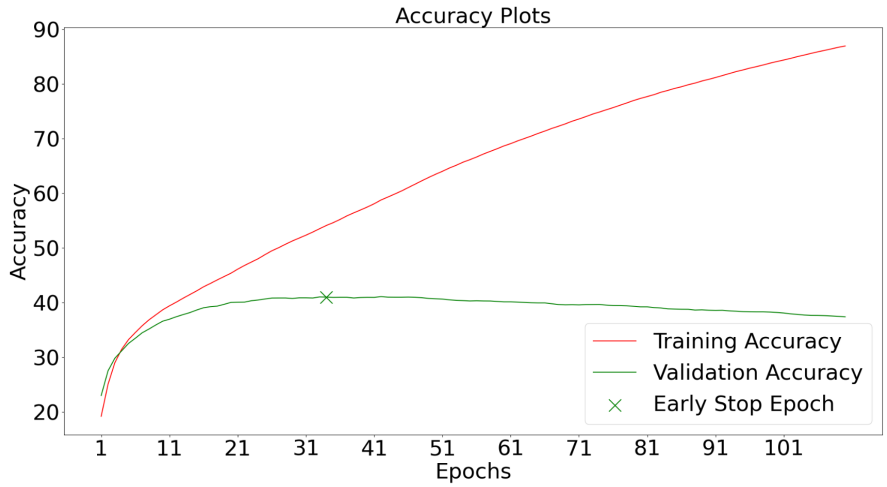
L2 Regularization with $\lambda = 10^{-4}$ Test Accuracy: 41.54 Test Loss: 1.7384873841356883

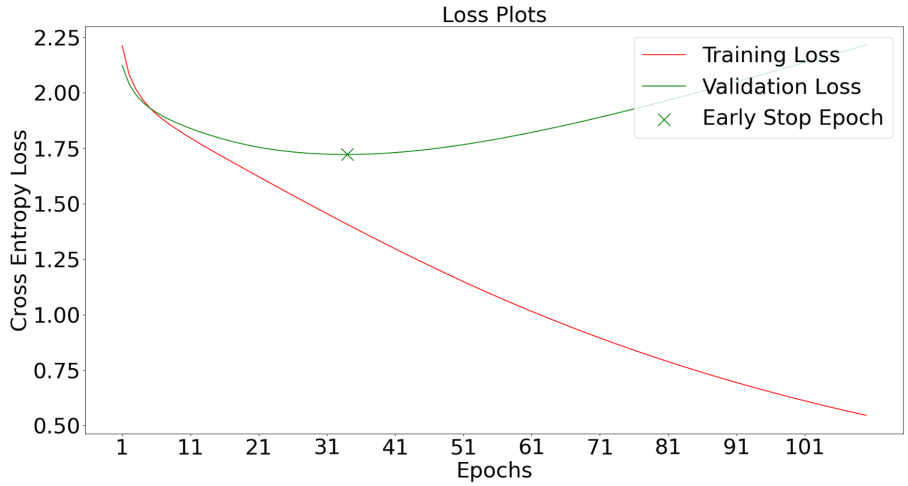
L2 Regularization with $\lambda = 10^{-2}$ Test Accuracy: 41.86 Test Loss: 1.7484002048629292

L1 Regularization with $\lambda = 10^{-4}$ Test Accuracy: 41.02 Test Loss: 1.7518425231996861

Training Procedure: Our training procedure is the same as what we have for momentum but our weight gradient equation changes as we now have an additional term for the gradient of the regularization term. For L1 regularization = $||w||$ the gradient gives us a constant which we get to be the regularization constant, λ. On the other hand for L2 regularization = $||w||^2$ the gradient comes out to be $\lambda|w|$.

Training vs Validation Plots with L2 regularization = 10^{-2}





Test accuracy, Observations and Inference:

With the best model using L1 regularization, we achieved a testing accuracy of 45.5% with $\lambda = 10^{-2}$ on our Network. Using the L2 regularization however, the best testing accuracy we got was 41.86% on our Network using $\lambda = 10^{-2}$. We found that L1 regularization worked better than L2 regularization, this indicates that our network's performance increases when the weight decay is harsher which is true in case of L1 as compared to L2 because L1 linearly decays all the weights whereas L2 decays bigger weights more whereas has a smaller decay on smaller weights. This claim is supported when we see the value of the regularization term that we choose. In terms of the value of λ we think that because L1 is performing better than L2 this would indicate that a harsher weight decay performs better for both L1 and L2 when $\lambda = 10^{-2}$as opposed to when $\lambda = 10^{-4}$. We found that L1 regularization worked better than L2 regularization.

Insights:

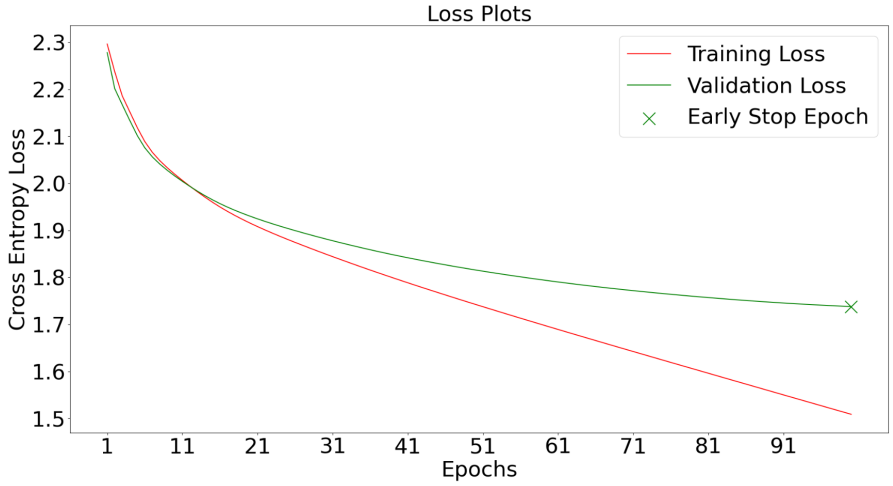
At first, our regularization did not improve our testing accuracy much, but we realized that in our backpropagation method we were normalizing not only the weight gradient but also the regularization term by the batch size which only made the decay negligible and that's why we think it didn't have any effect on our accuracy.

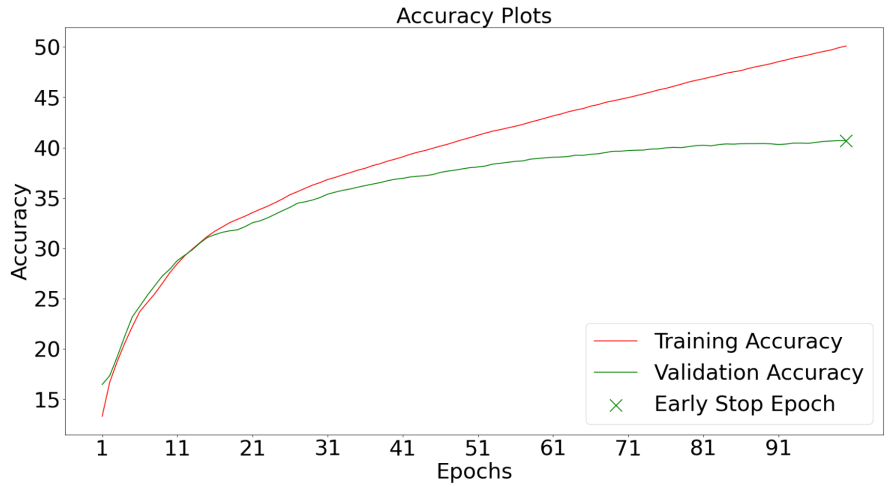
5 Activation Experiments

Activation Function: Sigmoid

Test Accuracy: 41.36 Test Loss: 1.7511202113810636

Training vs Validation Plots with Sigmoid Activation

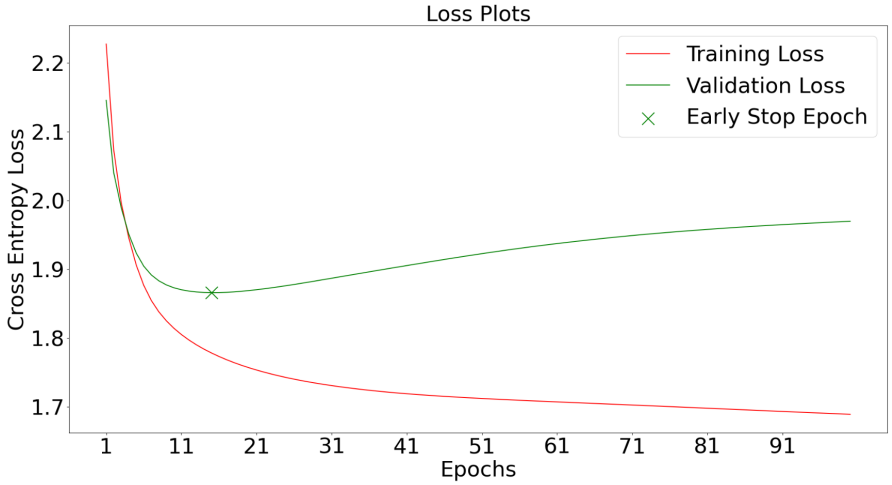


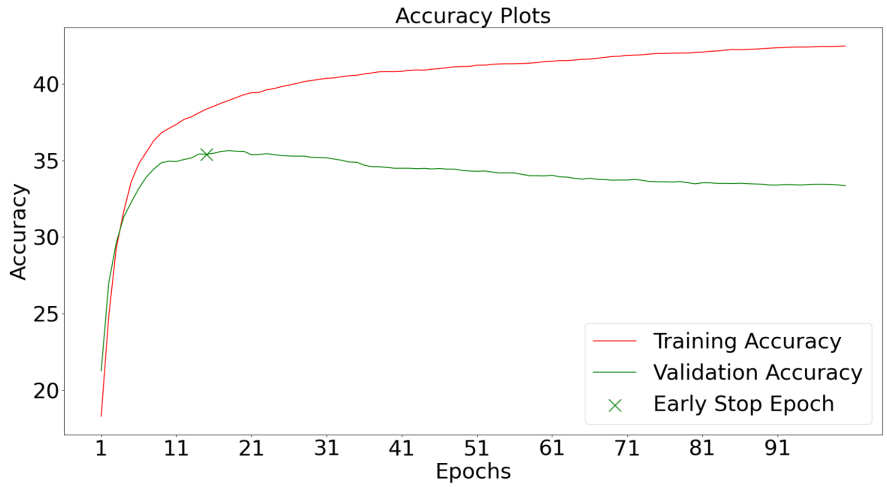


Activation Function: ReLU

Test Accuracy: 35.77 Test Loss: 1.852458151717401

Training vs Validation Plots with ReLU Activation





Observation:

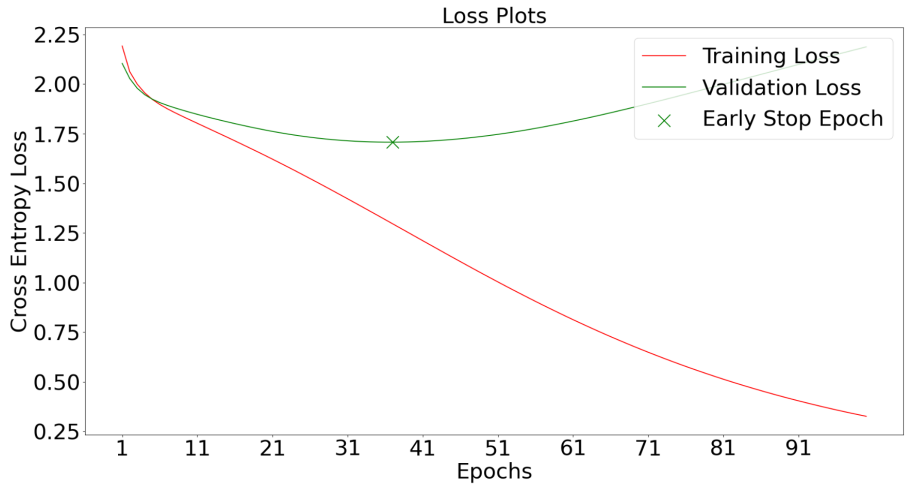
We know that the sigmoid and tanh function have a similar gradient which is why we think that the weight updates similarly give similar results as we got using tanh. Sigmoid works effectively for our network because we have only a single layer in our topology and if we were to have multiple hidden layers then the sigmoid accuracy would drop off compared to the tanh accuracy. With respect to the ReLU activation function we know this is better suited for deep neural networks and as we have only a single hidden layer it performs poorly compared to our tanh accuracy. When we were testing both of these activation functions an interesting observation was that the validation accuracy always seemed to be higher as compared to the testing accuracy.

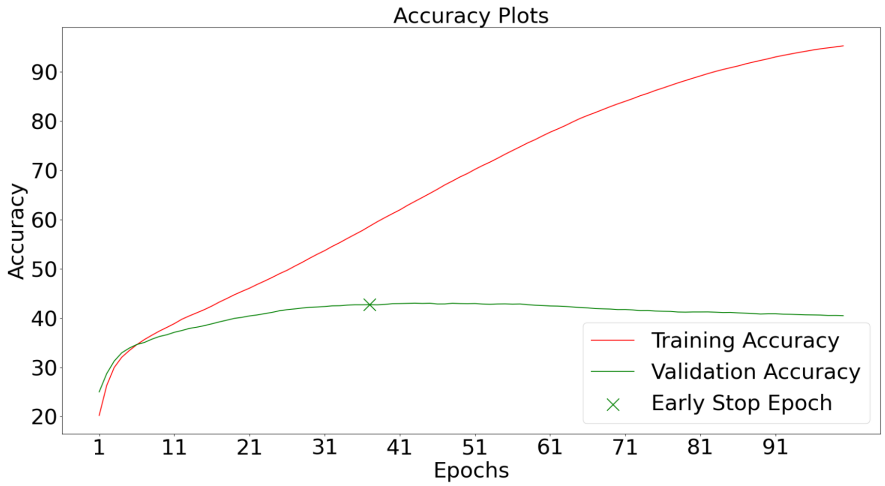
6 Network Topology Experiments

i. Doubling the number of hidden units.

As we used 128 units in the hidden layer we increased the number of hidden units to 256 units.
Test Accuracy: 43.89 Test Loss: 1.7260500217232653

Training vs Validation Plots with double hidden units

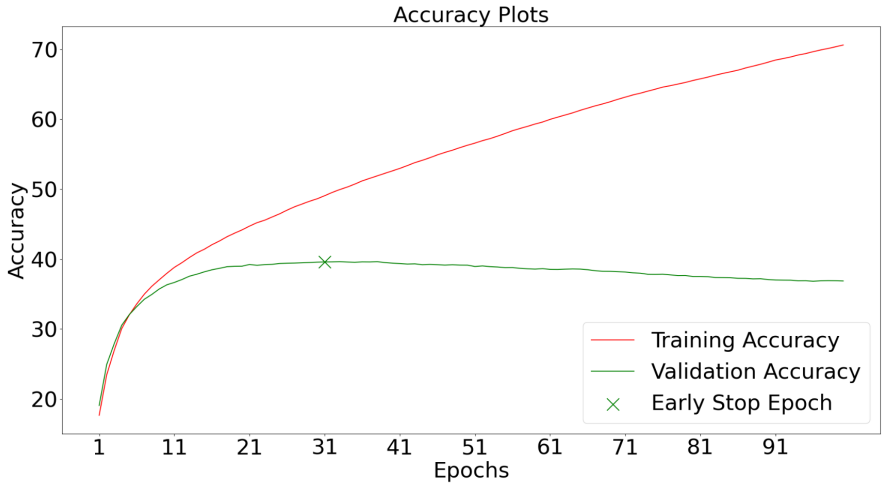


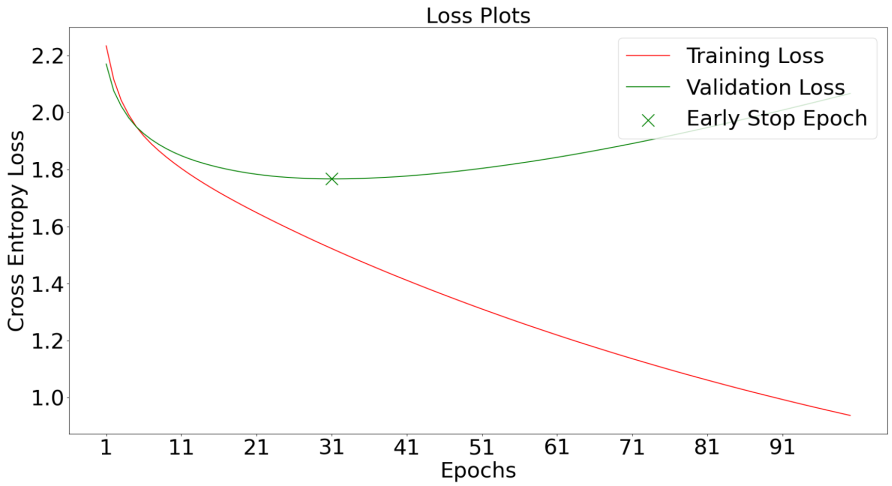


ii. Halving the number of hidden units.

As we used 128 units in the hidden layer for 2c, we decreased the number of hidden units to 64 units.
Test Accuracy: 39.36 Test Loss: 1.779612821007585

Training vs Validation Plots with half hidden units

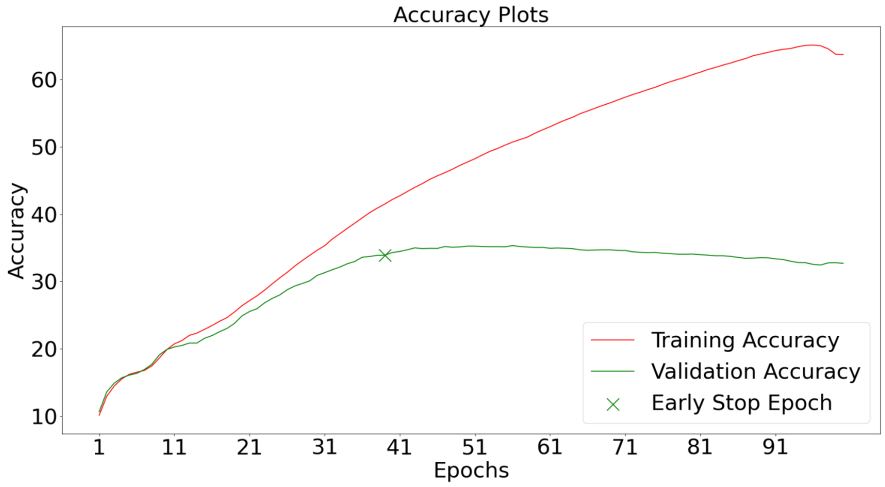


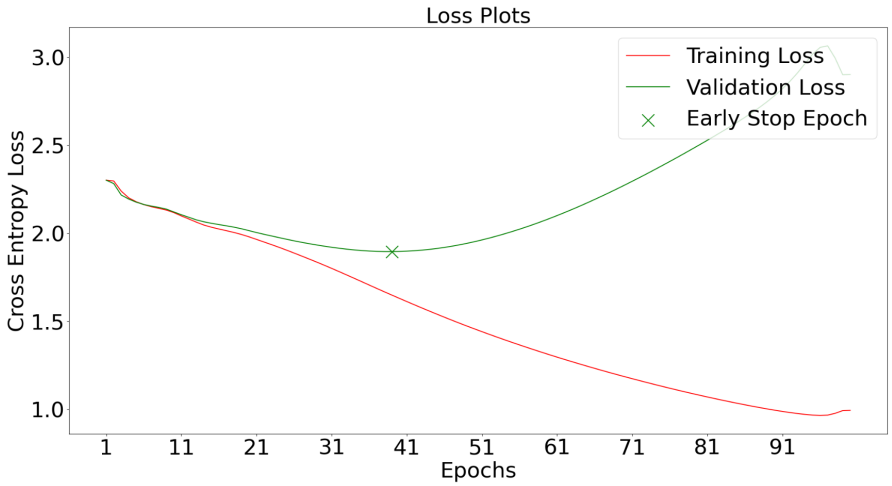


Observations:
We see that by doubling the number of hidden units our accuracy increases to around 43% from 41% whereas when we halve the number of hidden units the accuracy decreases to about 39%. This makes sense to us because we think that if we have more hidden units we are able to learn more complicated features of our training data as compared to when we half the number of hidden units we lose out on information because we reduce the number of features we train for.

iii. Doubling the number of hidden layers.
As we used 1 hidden layer for 2c, we increased the number of hidden layers to 2.
Test Accuracy: 34.28 Test Loss: 1.8928927833205647

Training vs Validation Plots with 2 hidden layers





Observation:
We were unable to understand why our accuracy drops when we increased the number of hidden layers to 2. We reduced the number of units to 64 each because we thought this would maintain the number of parameters based on the model we use in 2c. Our understanding was that when we increase the depth of our network it should ideally increase the accuracy but it turns out that our accuracy drops significantly to around 34%. For this reason we tried to test our data with 2 layers of 128 units which did bring our accuracy to about 39% which still performs worse than our single hidden layer network.

7 Team Contributions

For this assignment, our team primarily met in Geisel or on Zoom and worked collaboratively. If we met online, we used Zoom and Visual Studio Code “Live Share” to work in real-time with each other. When we met in Geisel, we made use of “Live Share” while also using the whiteboards to help with visualization of our Neural Network and matrix dimensions. Our team split up small portions of the coding, but worked together heavily on NeuralNet.py and main.py.

Raj wrote the util.py functions with Ramtin and configured the config (.yaml) files. Raj collaborated with Henry and Ramtin in order to complete the NeuralNet.py functions and set up the main.py file. He also was responsible for completing and testing the gradient.py file with Henry. Raj attended Office Hours and posted Private Piazza questions for the group in order to ensure our experiments with Momentum and Regularization worked as intended.

Henry was responsible for the maintenance and organization of the GitHub repository and ensured everyone’s code was merged correctly. Henry tested util.py functions and wrote the train.py functions. Henry collaborated with Raj and Ramtin in order to complete the NeuralNet.py functions and set up the main.py file. He was also responsible for testing the gradient.py functions worked as intended with Raj. Henry also attended Office Hours to ensure we had the correct understanding of Momentum and Regularization.

Ramtin was responsible for completing the util.py functions with Raj. Further, Ramtin collaborated with Henry and Ramtin in order to complete the NeuralNet.py functions and set up the main.py file. Ramtin tested the train.py functions and maintained main.py as we added new features to our Neural Network. Ramtin also attended Office Hours to get assistance for the group when we found ourselves stuck on a challenge.

Our group spent the most of our time working synchronously when coding, testing our functions, and performing the necessary experiments on our Network. We completed the report on a Zoom call where we discussed our overall findings and delegated different portions of the writing fairly among team members.

References

[1] Bishop, Christopher M (1995) *Neural networks for pattern recognition. Chapter 4, section 8.4*

[2] Cortell, Garry (2022), CSE151B lectures, University of California, San Diego, Fall 2022