Image Categorization with Multi-Layered Neural Networks

Ma, Junhua (Michael)
PID: A16299193
jum002@ucsd.edu

Lee, Elliot
PID: A15796673
eal001@ucsd.edu

Abstract

In this programming assignment we implemented a classifier for the CIFAR-10 image data set using multi-layer neural network, and tested different modifications to the network to analyze the effect of these modifications on the performance of the network. For the default network implementation we utilized momentum to improve the results and efficiency of the network in back propagation, and our network roughly met the expected accuracy at 42%. Using this network as the basis for comparison, we first tested L1, L2 regularization methods, which we realized improved performance with accuracy reaching 44% using L1 regularization at a smaller regularization constant of 10^{-4}. We also tested various activation functions like Sigmoid and RelU, and obtained notable improvements to accuracy with ReLU up to 47%. Finally, we tested various network topology by changing the size of the hidden layer as well as adding an additional hidden layer. We noticed that increased size of hidden layer leads to an increase in performance and we failed to notice any improvements from adding an additional hidden layer.

1 Data Loading

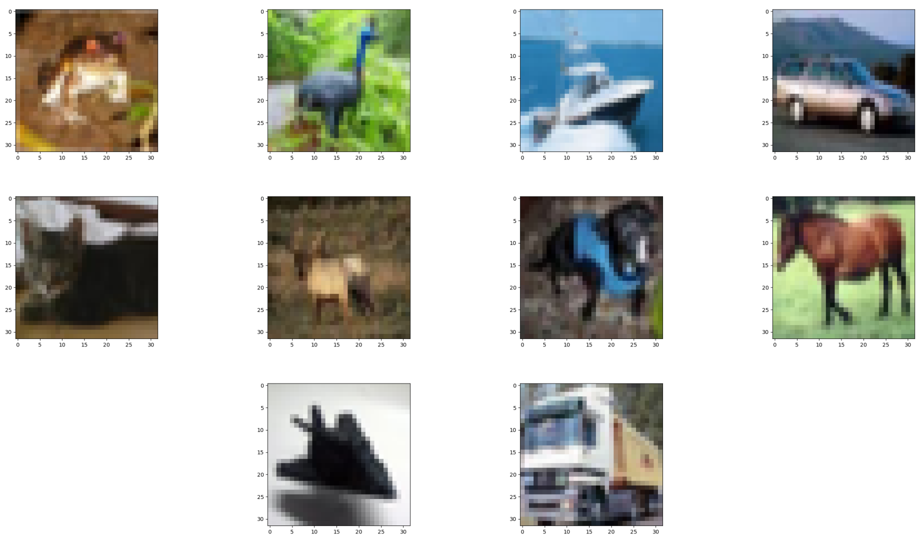


Figure 1: Image of Frog, Bird, Boat, Car, Cat, Deer, Dog, horse, Plane, Truck (in order left to right, top to bottom) in Training Set of CIFAR-10

This Programming assignment uses the data set CIFAR-10 to train a multi layered neural network to categorize images. The images are colored 32 x 32 images of 10 different categories. Because of their colorful nature, these images are separated out into 3 channels, one for red, one for green, and one for blue. They are then flattened into three 1024 long arrays, which are then concatenated into one 3072 long array. This was done so that the network can have a 1-dimensional array of information to pull and learn from. The data is then normalized across the image where each channel and example are normalized to have a mean 0 and standard deviation 1.

The distribution statistics of the data are listed below.

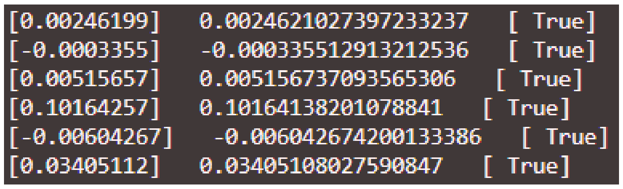
```
Training Set:
    Standard Deviation : 64.14303549406361
    Mean               : 120.76026919759114
Validation Set:
    Standard Deviation : 64.17542916489087
    Mean               : 120.49296309839318
Testing Set:
    Standard Deviation : 64.06097012299574
    Mean               : 121.52915475260417
```

2 Numerical Approximation of Gradients

After our initial implementation of back propagation, we checked our implementation using numeric approximation by calculating the gradient as follows:

$$\frac{d}{dw}E^n(w) = \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon} \tag{1}$$

where ϵ is selected to be a small constant 0.01. With this setup, we would expect the gradient calculated from back propagation to be accurate within a margin of 0.0001. To calculate the numerical approximation according to the formula, we first selected a couple of weights at random from different layers. For each weight, we calculate the gradient of the loss function with respect to the weight by first adding epsilon to the weight and forward the network to get the loss E_1, then we reset the weight, subtract epsilon, and forward the network again to obtain the loss E_2. With E_1 and E_2, we can calculate the numerical approximation of the gradient with respect to this particular randomly selected weight as $\frac{E_1 - E_2}{2\epsilon}$. The value obtained is then compared to the gradient dw obtained through back propagation, with the update rule $w = w - \alpha \cdot dw$, so w_{ij} is updated by adding the corresponding dw_{ij}. We printed the result for each of the weights and obtained the table as shown in **Figure 2**. Since the gradients calculated from back propagation all agree with the numerical approximation of the gradient for each weight, we can confirm that the back propagation - without momentum or regularization - is performing as expected.



```
[0.00246199]   0.0024621027397233237   [ True]
[-0.0003355]   -0.000335512913212536   [ True]
[0.00515657]   0.005156737093565306   [ True]
[0.10164257]   0.10164138201078841   [ True]
[-0.00604267]   -0.006042674200133386   [ True]
[0.03405112]   0.03405108027590847   [ True]
```

Figure 2: List of Numerical Gradient Approximations compared to the calculated Gradient in a Layer of the Neural Network. Left column is numeric approximation, middle column is gradient from back propagation, right column is whether the absolute difference is less than 0.0001. Row 1 to 2 are two random weights selected between input to hidden layer. Row 3 is a random weight between input bias and a hidden unit. Row 4 is a random weight between hidden layer bias to a output layer unit. Row 5 and 6 are two random weights selected between hidden to output layer. The absolute difference values from top to bottom are: 0.000001, 0.00000001, 0.0000002, 0.000001, 0.000000001, 0.0000001.

3 Neural Network with Momentum

The first learning tool that we experimented with was momentum. This is when we compare the current loss gradient to how lour loss gradient has looked previously. We use a velocity or momentum term to accelerate the changes to the weights.

3.1 Initial Implementation

Our initial implementation of the multilayered gradient descent involved keeping track of the hidden layer and the output layer. For every layer, the weights are updated using a delta term that is calculated by comparing the desired value to our network's current activation. For the output layer our deltas are simply:

$$\delta = target - activation \tag{2}$$

However for the next layers back, the deltas are calculated as:

$$\delta = g'(a) \sum^{k} \delta_k w_k \tag{3}$$

where k is in the set of all outgoing weights from the current activation. The two deltas are calculated differently because at the output layer we want to update the incoming weights based on their final output. However, within hidden layers, we want to adjust incoming weights based on their final output, and the effect that they have on future layers. This is why we account for the $\sum k\delta_k w_k$ in the activation deltas. We then used these calculated deltas in our calculation of the cross entropy loss gradient as

$$\frac{\partial E}{\partial w} = X\delta \tag{4}$$

where X is the list of each activation input to a layer and δ is the matrix of corresponding deltas to the output nodes of this layer. Then we added a momentum term that iterated off of itself to allow for faster updating of weight. Initially momentum for any weight is 0, but on every iteration we store the difference between the current momentum and the current gradient.

$$M = \gamma M - \frac{\partial E}{\partial w} \tag{5}$$

here γ is a small number close to 1 and M is a matrix to represent momentum, with the same shape as as the weights for each layer. Finally, we updated the existing layer's weight matrix using a simple update rule.

$$W = W - \alpha M \tag{6}$$

Here W is the weight matrix corresponding to this layer with N (input) rows, and M (output) columns and α is a small constant called the learning rate. This implementation failed to learn the data. The loss plots indicated that it wasn't learning as the loss levels stayed fairly flat over each implementation. The final result for this iteration of our neural network was about 20% test accuracy.

3.2 Next Iteration

In order to improve accuracy, we iterated on our gradient calculation and our gradient descent formula. One edit that that we made was to normalize the gradient. Because we are performing gradient descent on a mini-batch , there are multiple examples being evaluated on one iteration. When the number of examples increase, so too does the term $X\delta$. We want to normalize these sums that we are computing by dividing the number of examples in the batch.

$$\frac{\partial E}{\partial w} = X\delta/(batchsize) \tag{7}$$

The other change was adjusting the velocity formula. Previously, we applied a learning rate to velocity, but the network learns better if velocity is unrestricted by the learning rate. Additionally the model fits the data better if the gradient update to velocity is multiplied by $1 - \gamma$. So our new calculation for momentum became:

$$M = \gamma M - (1-\gamma)\alpha \frac{\partial E}{\partial w} \tag{8}$$

and our update to the weights now is:

$$W = W - M \tag{9}$$

The resulting accuracy with this algorithm consistently hit 42% accuracy over multiple iterations of this experiment.

The hyper-parameters used for this network are a learning rate of 0.01, batch size of 128, momentum gamma of 0.9, an activation function of tanh, and trained over 100 epochs with early stopping. Additionally the network architecture is a 3072 input, 128 activation hidden layer, and a 10 category output.

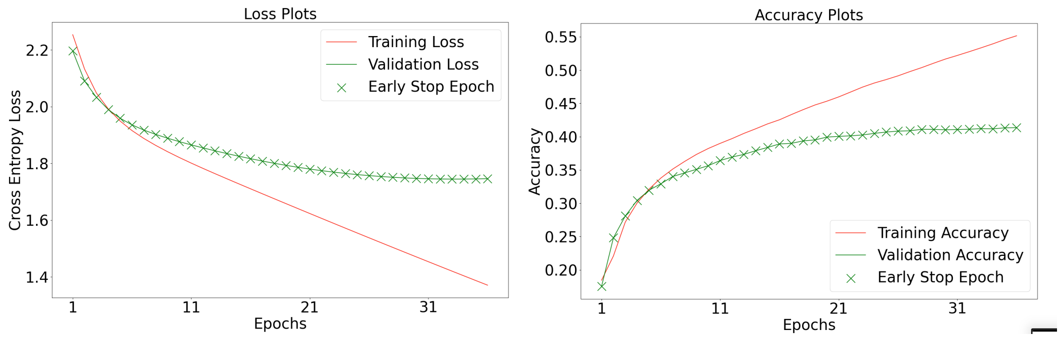


Figure 3: Loss and Accuracy of Network Categorization for part 2c

4 Regularization Experiments

Regularization is a useful tool for dealing with over-fitting and improving the generalization of the network. By penalizing the weights and make them smaller, we can reduce the complexity of the model and help minimize the loss given by the following equation:

$$J = E + \lambda C \tag{10}$$

where E is the error and C is the complexity. There are two types of regularization: the L2 regularization penalize bigger weights more than smaller weights while L1 regularization penalizes big weights and small weights equally. Our experiments will test both L2 and L1 regularization, and we will also try different regularization constants 0.01 and 0.0001 to modify the amount of penalization. The formula for L2 and L1 regularization is given as:

$$\text{L2 Regularization: Minimize } C = ||w||^2$$
$$\text{L1 Regularization: Minimize } C = ||w||$$

To apply regularization to back propagation, we take the derivative of complexity with respect to the weight and add the resulting term to our weight update rule to penalize the weight values. So for L2 regularization we subtract an additional λw term in the weight update rule, and for L1 regularization we subtract an additional λ term from the weight update rule.

4.1 L2 Regularization

We tested the effect of adding L2 regularization to the network and tested 2 regularization constants as 0.01 and 0.0001. For L2 regularization with regularization constant 0.01, the resulting performance of the network is shown in Figure 3. For L2 regularization with regularization constant

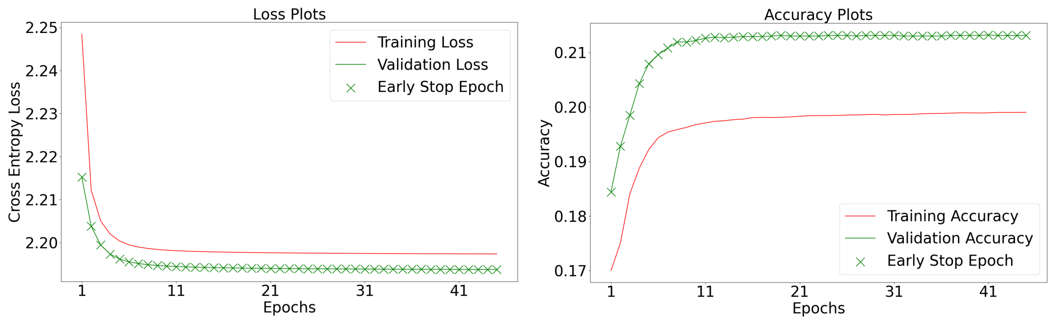


Figure 4: Loss and Accuracy of Network with L2 Regularization at $\lambda = 0.01$ (Part 2d)

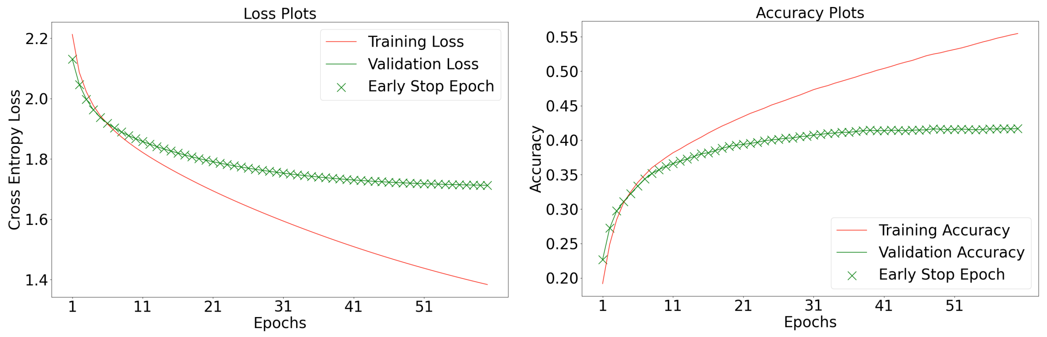


Figure 5: Loss and Accuracy of Network with L2 Regularization at $\lambda = 0.0001$ (Part 2d)

0.0001, the resulting performance of the network is shown in **Figure 5**. Comparing the performance of network with L2 regularization at a regularization constant of 0.01 and 0.0001, we can see from the performance plot that the performance is much better with regularization constant of 0.0001, at which the accuracy reached about 41%. However, compared to the network performance without any regularization, there's no clear improvement or very little improvement to network performance. For the improved performance with smaller regularization constant, our reasoning is that it's not beneficial to penalize the weights too hard as weights too small makes it difficult for the network to learn different features. And for the general lack of improvement to performance with L2 regularization,

4.2 L1 Regularization

We also tested the effect of adding L1 regularization to the network at regularization constant of 0.0001, and the resulting performance of the network is shown by **Figure 6**. Using L1 regularization,

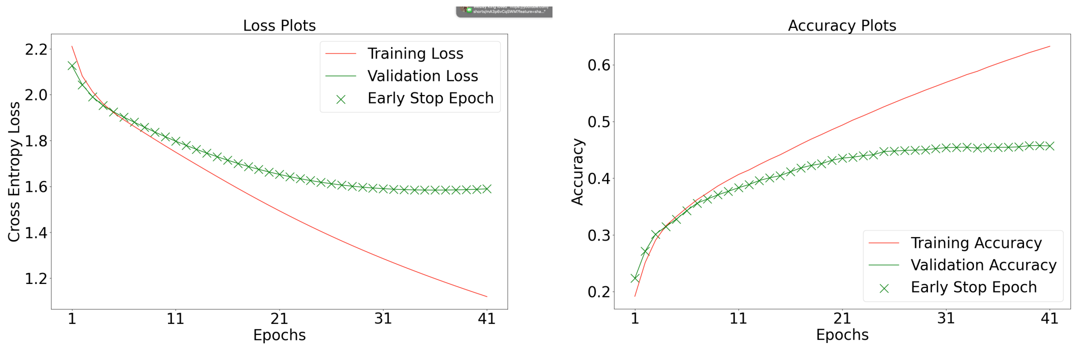


Figure 6: Loss and Accuracy of Network with L1 Regularization at $\lambda = 0.0001$ (Part 2d)

we see a notable improvement to the performance of the network, with accuracy went above 44%. This improvement shows the effect of regularization to help prevent over-fitting, which leads to improved generalization of the network from learning and leads to improved validation accuracy.

5 Activation Experiments

Different activation functions can provide different node activation on each of the layers. We experimented with the same hyper-parameters that worked with our momentum experiments and adjusted the hidden layer activation function. Certain functions like the tanh and the sigmoid activation may suffer from issues like a vanishing gradient, because the derivative taken during backward activation may be very small, and make it tough to change the weights on earlier stages of the network.

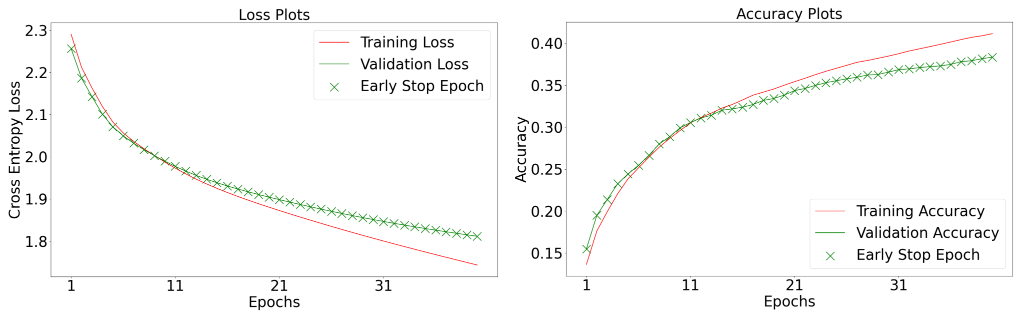


Figure 7: Loss and Accuracy of Network Categorization with Sigmoid activation (Part 2e)

The performance plots of the network using the sigmoid activation function is shown by **Figure 7**, and that of the network using ReLU activation function is shown by **Figure 8**. Using the activation of sigmoid we achieved an accuracy of 37%. However ReLU does not suffer from the same issues, and thus it performs better. Using ReLU we achieved a higher accuracy of nearly 47%.

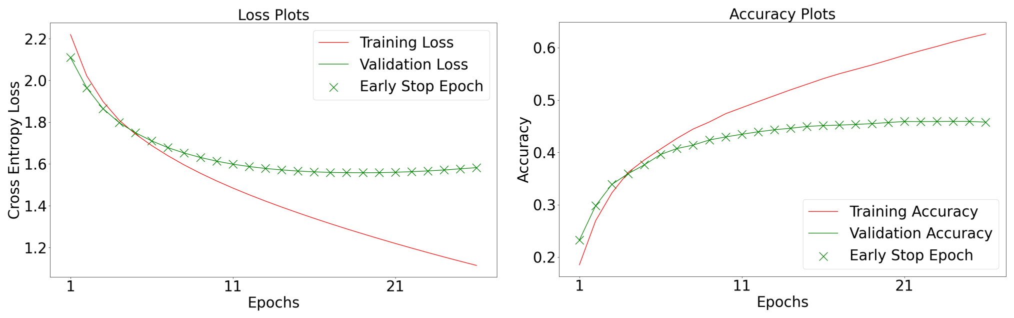


Figure 8: Loss and Accuracy of Network Categorization with ReLU activation (Part 2e)

6 Network Topology Experiments

Different network topology like changing the number of the hidden layers or the number of nodes in the hidden layer can also impact the performance of the network.

6.1 Changing Number of Units in Hidden Layer

We tested how the network performs when the number of units in the hidden layer is multiplied by 2 and divided by 2. Since the model we used to compare has 128 hidden units, we tested the performance of the model when there are 256 hidden units and when there are 64 hidden units.

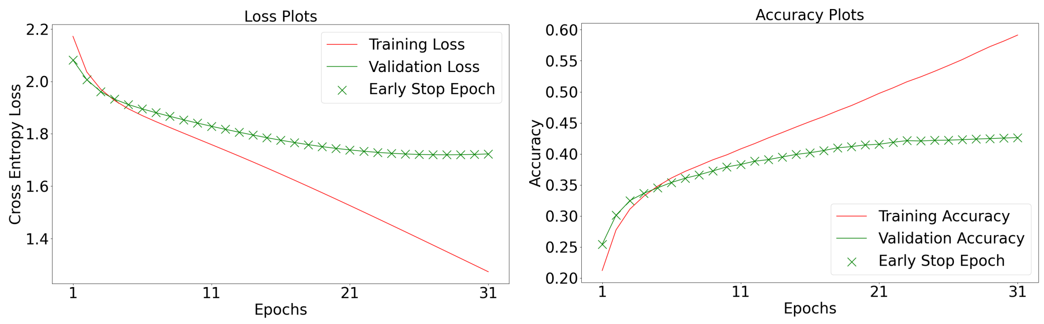


Figure 9: Loss and Accuracy of Network with 256 hidden layer units (Part 2f(i))

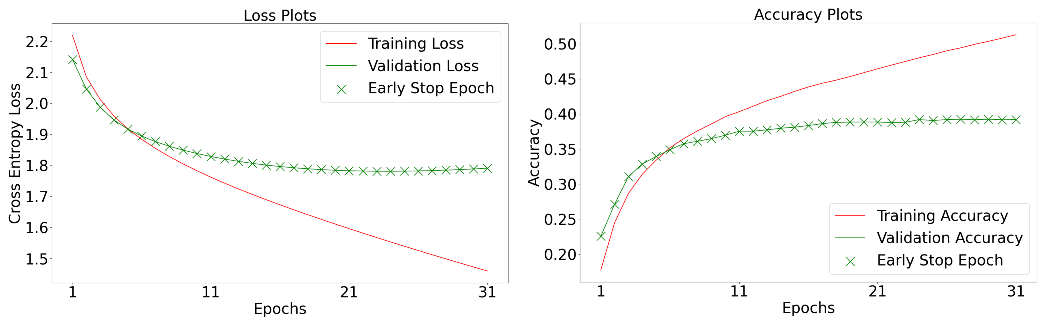


Figure 10: Loss and Accuracy of Network with 64 hidden layer units (Part 2f(i))

Since the number of weights are also associated with the number of hidden units, increased number of hidden units also leads to increased number of weights in the network. We expected that increased number of hidden units should improve the performance of the network since since the network can learn more features from the images and thus can generalize better to different images, and thus improve the overall performance in terms of increasing the accuracy and decreasing the loss. After running the tests, the performance of the network with 256 hidden layer units are shown in **Figure 9**, the network with 64 hidden layer units are shown in **Figure 10**. From the performance plots, we can see that the network with 256 hidden layer units is slightly better than the default network with 128 hidden units, reaching an accuracy of over 42%. On the other hand, we can see that the network with only 64 hidden layer units performed worse than the default network as the accuracy is below 40%. Therefore, this result agrees with our expectation and confirms the benefit of additional hidden layer units at improving the accuracy of the network. However, the increased number of hidden layer units does lead to a slight increase in training time.

6.2 Changing Number of Hidden Layers From 1 to 2

Next, we tested how the network would perform with an architecture with 2 hidden layers, but with approximately the same amount of parameters. Our best neural network architecture was an input of 3072, a hidden layer of 256 hidden units, and an output of 10. In total this was 786688 weights and biases. Applying this total amount of weight to two hidden layers of equal size, we found that each of the two hidden layers should have 237 hidden units. Training on this double hidden layer network architecture performed worse than the initial 1 hidden layer [3072, 256, 10] network architecture. It had about a 39% test accuracy. We believe that this is due to the network learning more abstract structures in the images, but over-fitting them. The data input is only 3072 large, so it is possible that less layers are really needed to abstract the image.The features learned at the next layer may have learned to be too specific to the training data. So, despite being a deeper network, the accuracy is lower than shallower ones.

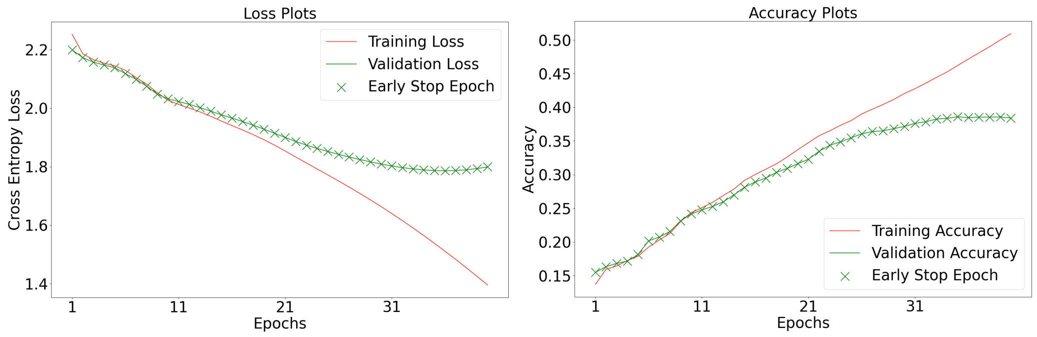


Figure 11: Loss and Accuracy of Network Categorization with 2 Hidden Layers (Part 2f(ii))

7 Team Contributions

7.1 Michael's Contribution

I implemented most of the Activation class, which includes forward activation functions and backward gradient functions. I implemented the initial Layer class and contributed to testing and fixing the backward function throughout the project. I implemented the back propagation function of the Neural Network class. I contributed to programming the function for numerical approximation of gradients. I contributed to all of the experiments in this assignment, from preparing config files to discussing and analyzing the results. Finally, I contributed to writing the abstract as well as sections 2, 4, and 6 of this report.

7.2 Elliot's Contribution

I implemented data loading and normalization, and contributed to the initial implementation of our forward and backward functions. I contributed to the gradient test and made numerous iterations on layer class's backward and forward function, the neural network class's forward function, and the gradient numerical approximation. All the experiments and testing were co-conducted by me. I performed some hyper-parameter testing, and contributed to the report by writing sections 1, 3, 5, and 7 of this report.

8 Related Work

The following materials and resources are referenced during the assignment:

CIFAR-10 Website: https://www.cs.toronto.edu/ kriz/cifar.html – understanding the data format and what is contained within it
Piazza Post 495 – understanding what reasonable accuracy is
Piazza Post 503 – directions on tuning hyper-parameters
Piazza Post 534 – understanding reasonable accuracy of adding an extra hidden layer
Improve Generalization Lecture Slides – understand Regularization formulas and ideas
Trick of the Trades Lecture Slides – understand momentum and the momentum equation and mini-batch SGD.
Shubham's Office Hour – learnt about alternative form of the momentum formula: $v(t) = \beta \cdot v(t-1) + (1 - \beta) \cdot \frac{\partial E}{\partial w}$