CSE 151B Homework 2 Programming Assignment

Mitchell Zhang
miz134@ucsd.edu

Jun (Moses) Oh
j6oh@ucsd.edu

Abstract

In our project, we classify digits cropped from the SVHN dataset. We created a neural network that does a forward pass and then backpropagation on the network. Our hyperparameters were set using a config.yaml file, which included number of layers, hidden units, activation function, learning rate, batch size, epoch size, early stopping, regularization constant, and momentum. We used one-fold cross validation and mini-batch stochastic gradient descent. We used cross-entropy loss function as our error/objective function. Included in our tests also were using different activation functions, between sigmoid, tanh, and ReLU, and using regularization and momentum or not. Our initial test accuracy (using tanh and not using regularization or momentum) was 74.15%. Using momentum decreased our test accuracy but increased the training speed (if using early stopping). Our test accuracy with regularization peaked at 77.8%. We found that the best activation function to use in our neural network was ReLU, peaking at a test accuracy of 83.4%. We also tested with changing number of hidden layers and units.

1 Introduction

In this project, we implemented a neural network to classify images of digits. We would use a mini-batch to train our model using a forward pass and then backpropagation to change its weights. After running this mini-batch stochastic gradient descent across the whole training example, we would test loss and accuracy on a validation set(80-20 training vs validation data set). Below shows graphs of accuracy and loss for different parameters as epochs increase. Inital test accuracy without momentum or regularization using tanh was 71-74%, peaking at 74.15%. Our hyperparameters consisted of layer specifications, activation function, learning rate, batch size, epoch size, early stopping epoch, regularization constant, and momentum gamma. We found that momentum primarily increased speed, rather than accuracy. Learning rate had to be tweaked depending on the activation function used, but generally, ReLU activation function provided the best accuracy. Our accuracy peaked at 83.4% on a test set separate from that being used to train or validate. Regularization also increased our accuracy, depending on the constant used. Decreasing number of hidden units decreased accuracy while increasing slightly increased accuracy. Increasing number of hidden layers noticeably increased accuracy, up to 80.6% with standard configurations (without regularization and using tanh).

2 Dataset

The dataset that we used came from the SHVN dataset. These consists of street address numbers, with each digit in the number cropped as a single data example. There are about 73000 training examples and 26000 testing examples. We split the training examples into 80-20 training and validation sets. We used one-fold cross-validation when training our model. We normalized the x values by z-scoring it. We used the average image across all examples and subtracted it and divided by the mean and standard deviation respectively. The resulting input would have a mean of 0 and standard deviation of 1. We used these same average mean and standard deviation to normalize the validation

and testing datasets as well. We also converted the labels to a one-hot encoding of the digits. All the labels were the digits between 0 and 9. Prior to splitting and training out dataset, we shuffled the dataset.

3 Backpropagation/Momentum

3.1 Backpropagation

Our neural network was implemented based off a configuration file, which specified the layers. The default layer specification (and those listed below unless otherwise specified) was [1024, 128, 10]. We also used mini-batch sizes of 128 for all tests unless otherwise specified. Our implemented consisted of a Layer classes, with Activation classes in between each Layer. The Layer class would calculate the forward and backward passes, while the Activation class would calculate the activation function and its derivatives for each forward and backward pass. We used mini-batch stochastic gradient descent, using each batch to call the model's forward pass and backpropagation to change weights accordingly. We had an input to hidden layer, and hidden to output layer.

To calculate probability, we used the softmax function, stabilized for large numbers:

$$\hat{y}_i = \frac{e^{a_i - max(a)}}{\sum_{i=1} e^{a_i - max(a)}} \tag{1}$$

To calculate loss, we used the cross-entropy loss function, averaged across all examples:

$$E = -\frac{1}{n} \sum_n \sum_{k=1}^{c} y_k^n ln(\hat{y}_k^n) \tag{2}$$

To calculate the change in weights, we used the formula

$$w_{jk} = w_{jk} + \alpha \sum_n \delta_k^n z_j^n$$

For the output layer,

$$\delta_j = (t_j - y_j)$$

where t is the target and y is the predicted (output) of the model.

For the hidden layer,

$$\delta_j = g'(a_j) \sum_k \delta_k w_{jk}$$

where g'(a) is the derivative of the activation function

We used the vectorize computations for these formulas such as

$$W_{jk} = W_{jk} + \alpha Z^T \Delta_k$$

$$\Delta_j = g'(A) \times (\Delta_k W_{jk}^T)$$

Below is the accuracy and loss plot for our initial model test, with default configurations. This implementation does not have momentum or early stopping, uses an activation function of tanh, a learning rate of 0.005. The number of epochs was increased to 150 as we observed slightly higher accuracy (72-73% -¿ 74.2%) with larger epoch sizes.

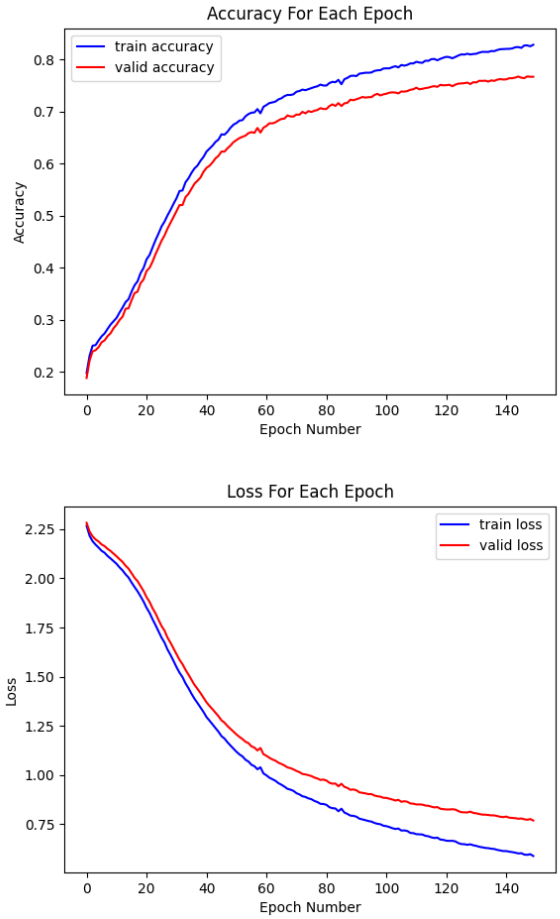

Figure 1: Activation = tanh, Epoch = 150, Learning Rate = 0.005, Final Accuracy = 74.2%

Numerical Approximation is calculated using this formula:

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

We used an epsilon of $\epsilon = 10^{-2}$. The differences calculated manually and with our program agreed within 10^{-6}

Weight	Approximation	Backpropagation	Difference
Output Bias	0.017383262665537913	-0.017382897982463008	3.64683075e-7
Hidden Bias	-0.0024599363956046005	0.0024600303908633558	-9.38852588e-8
Hidden-Output W1	0.040565235790857024	-0.04056585405246888	-6.18261612e-7
Hidden-Output W2	-0.08428286597483847	0.08428670529482442	-3.83931e-6
Input-Hidden W1	-0.0010690676859992365	0.0010690719483142485	-4.26231501e-9
Input-Hidden W2	0.007867567443531343	-0.007868651725138408	-1.08428e-6

Note: Our numerical approximation vs backpropagation values have opposite values because we update rule for weights is w = w + dw rather than w = w - dw. This simply a difference in implementation. The difference is calculated by taking the absolute value of both and then subtracting.

H-O refers to hidden to output weight

I-H refers to input to hidden weight

3.2 Momentum

To train our model, we used mini-batch stochastic gradient descent. We primarily tested using a batch size of 128 units, meaning we would do forward and backpropagation using 128 examples at a time, iterating through the training set with 128 examples at a time for a single epoch. We implemented simple momentum, with the choice of the momentum gamma in in config.yaml as a hyperparameter. We noticed that momentum did not necessarily increase accuracy, even decreasing it at times. However, it did noticably increase runtime, as we had also implemented early stopping. Our criteria for early stopping was not having a decrease in loss for x number of epochs, where x was equal to a configuration we set in config.yaml as another hyperparameter. Primarily, we chose an early stop epoch of 5 for our methods. While lower early stop epoch thresholds could decrease accuracy occasionally, higher thresholds did not increase accuracy. The model that had the least amount of loss on the validation set was saved to be used to test on the test set. Our momentum was added to the change in weight variable, in which the momentum consisted of the previous change in weight, multiplied by the momentum gamma set in the configuration file. We discovered some overfitting in general, especially as epoch number increased, despite implementing early stopping. However, we deemed that this much overfitting was acceptable, since validation loss was still decreasing, albeit only slightly.

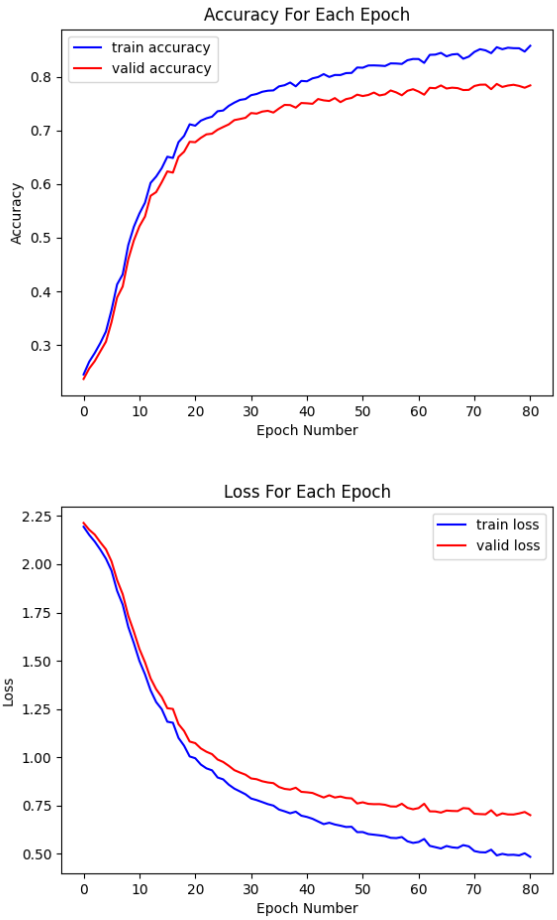

Figure 2: Activation = tanh, Epoch = 150, Learning Rate = 0.01, Early Stop = 5, Momentum = 0.4, Final Accuracy = 76.1%

We tested using multiple momentum gamma rates, including: {0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}. We noticed that the lower the momentum gamma, the longer number of epochs it would take. However, the accuracy would be much more consistent, matching rates to without momentum at around 71-74%. Higher momentum gamma rates, such as 0.9, caused the accuracy to fluctuate

much more widely and early stopping would be triggered early at around epoch 30, but the accuracy would be around 63-68%. In general, for highest accuracy, we found that using a low momentum gamma would be best, while for highest speed with early stopping, a high momentum gamma would be best. However, a too low momentum gamma would also decrease final accuracy with certain configurations (such as using ReLU).

4 Regularization

We utilized L2 Regularization in order to prevent possible over-fitting. We added a regularization term to weight update rule. We also trained 10 percent more epochs for regularization which totals to 110 epochs. We first tested the accuracy of the network without regularization 73.1% vs with regularization 74.2%. We can see an incremental increase in accuracy with regularization. Then we tested two different regularization coefficients - 1e-3 vs 1e-6. We see an accuracy of 73.1% for lambda set to 1e-6 vs an accuracy of 77.8% for lambda set to 1e-3. We can clearly see that a smaller regularization coefficient results in the highest accuracy score. We also tried out using L1 Regularization and saw an accuracy of 77.1%, a 3% increase from using L2 regularization. This may be due to the fact that L2 regularization usually works better for correlated inputs since it puts even weight on each, while L1 regularization takes a random one.

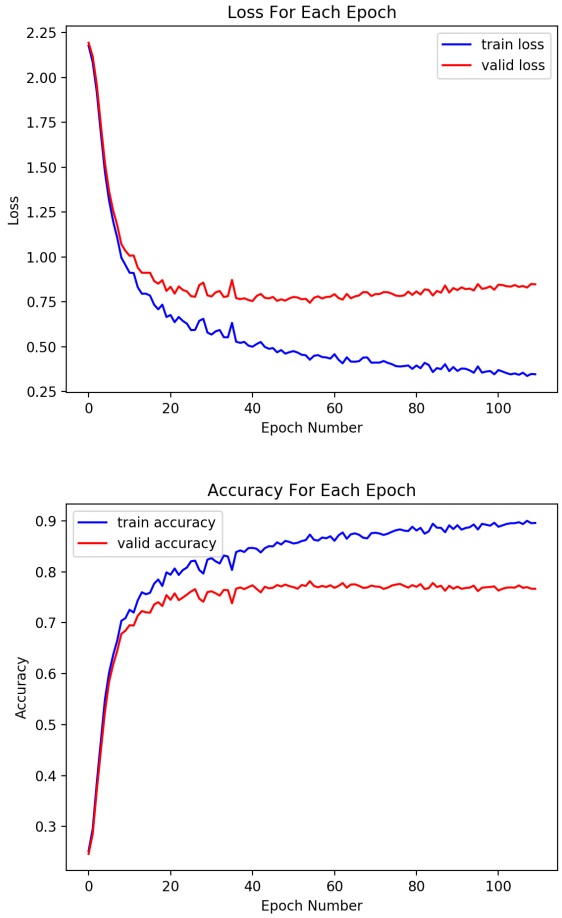

Figure 3: Without Regularization : Activation = tanh, Epoch = 100, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 73.1%

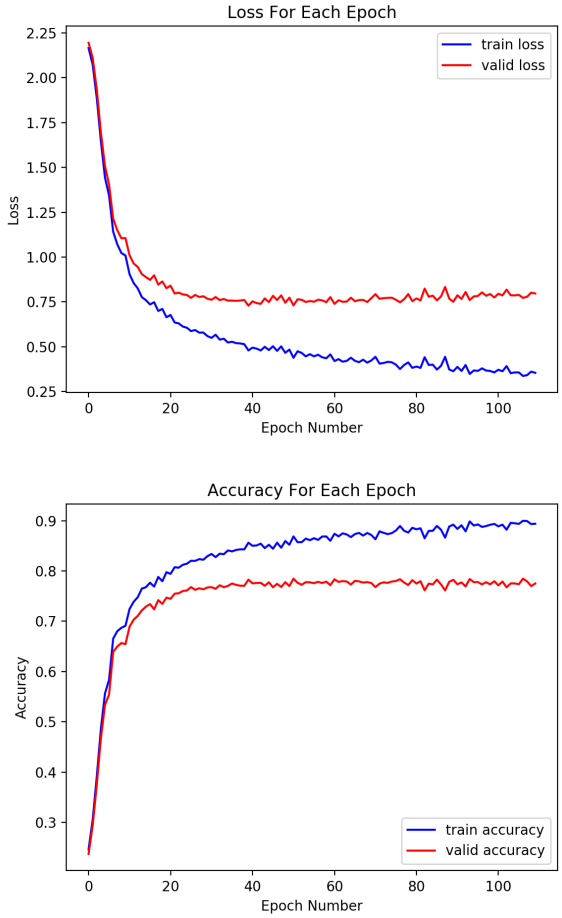

Figure 4: With Regularization : Activation = tanh, Epoch = 110, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 74.2%

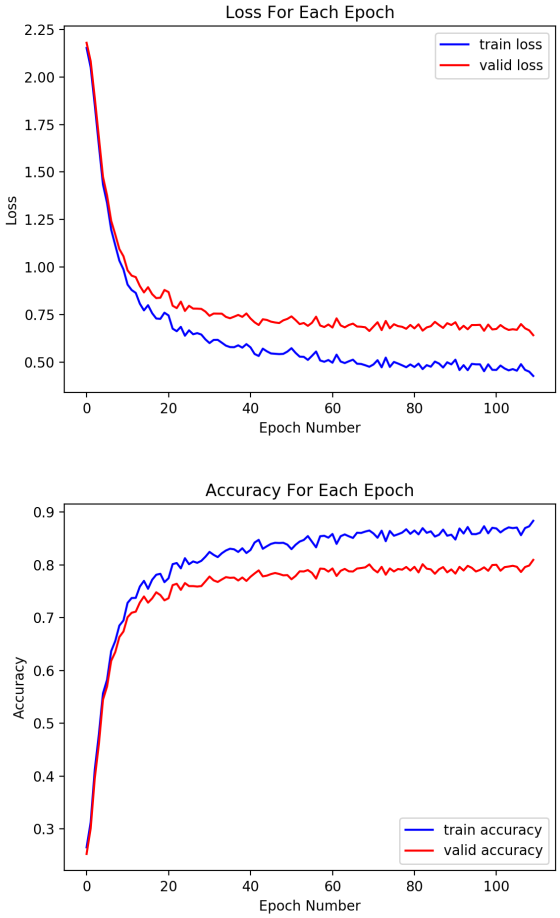

Figure 5: With Regularization 1e-3: Activation = tanh, Epoch = 110, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 77.8%

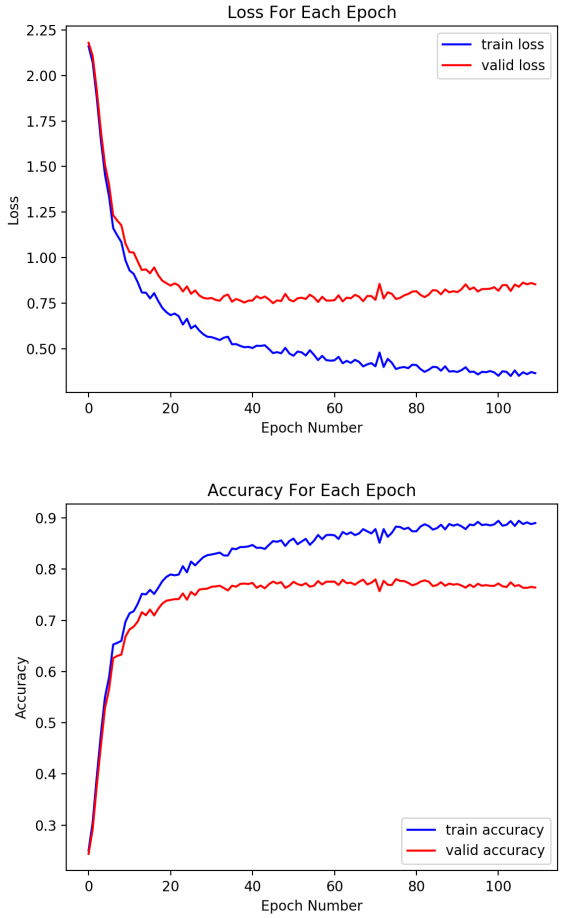

Figure 6: With Regularization 1e-6: Activation = tanh, Epoch = 110, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 73.1%

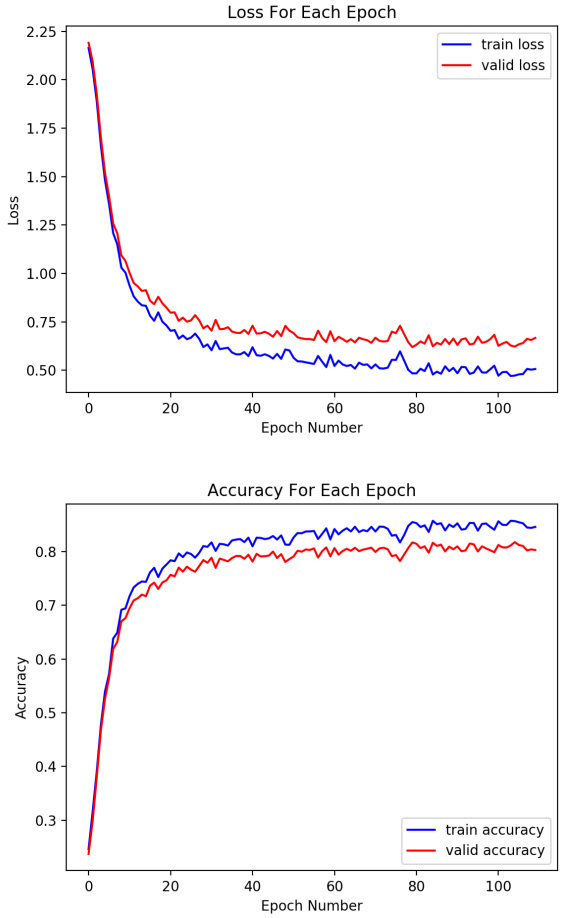

Figure 7: With L1 Regularization: Activation = tanh, Epoch = 110, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 77.1%

5 Activations

For activation functions, we had implementations for: sigmoid, tanh, and ReLU. We found that the worst performance was sigmoid at 74.2%, and the best performance was ReLU at 83.4%. For Sigmoid, we found that the optimal learning rate and momentum gamma was 0.01 and 0.4 respectively, matching similarly with tanh. For ReLU, we found that lowering learning rate had better results. Higher learning rates (such as 0.01) would cause loss to fluctuate heavily and early stopping would occasionally be triggered early, causing a low accuracy on the test set. A low learning rate (such as 0.001) would cause the training set to heavily overfit and lower accuracy. A higher/lower (0.8/0.4) momentum gamma also caused final accuracy to decrease. The difference in sigmoid and tanh as an activation function could be attributed to sigmoid returning results between 0 and 1, while tanh returns outputs between -1 and 1. ReLU shows the best accuracies as compared to both sigmoid and tanh. While sigmoid and tanh have similar functions and derivatives (relatively), ReLU has much different functions. For example, it's derivative is either 1 or 0, a stark contrast to sigmoid and tanh. Moreover, all negative values are snapped to 0 for its activation function. It can also provide larger values than 1.

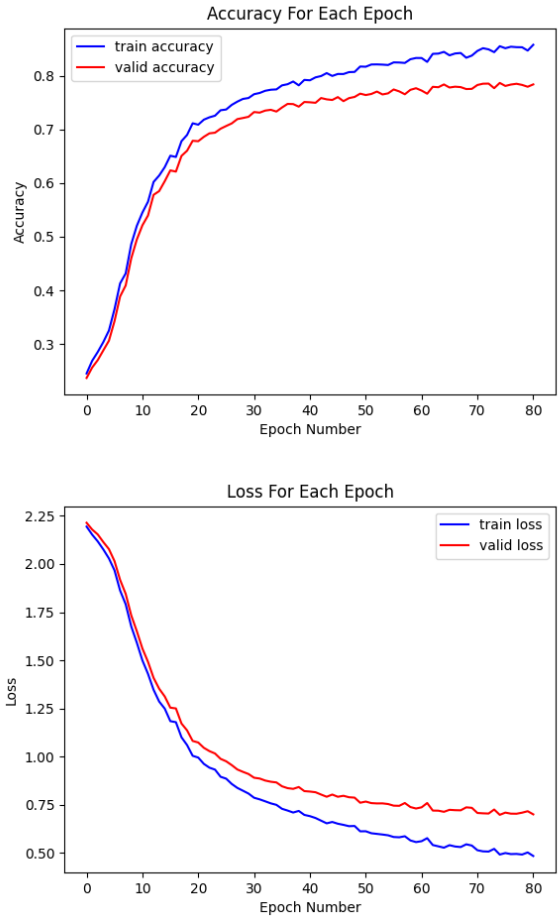

Figure 8: Activation = sigmoid, Epoch = 150, Learning Rate = 0.01, Early Stop = 5, Momentum = 0.4, Final Accuracy = 74.2%

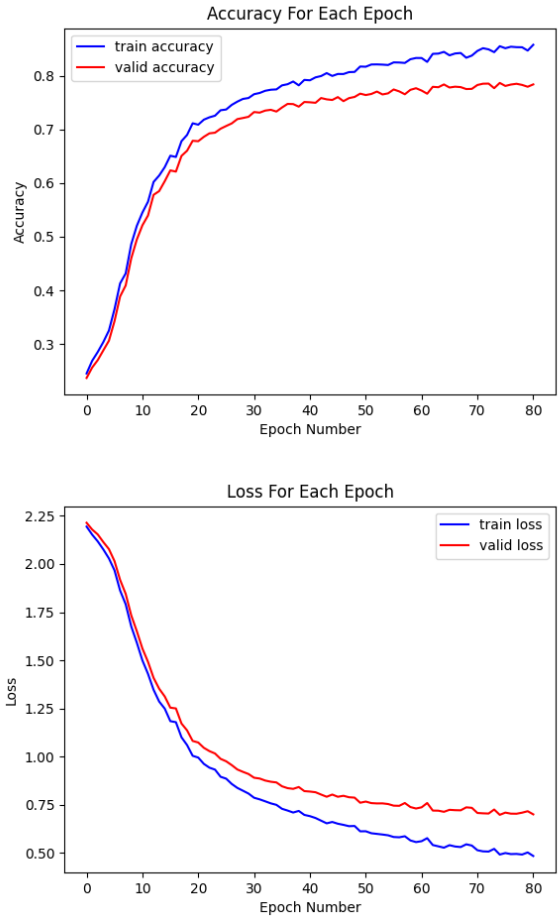

Figure 9: Activation = tanh, Epoch = 150, Learning Rate = 0.01, Early Stop = 5, Momentum = 0.4, Final Accuracy = 76.1%

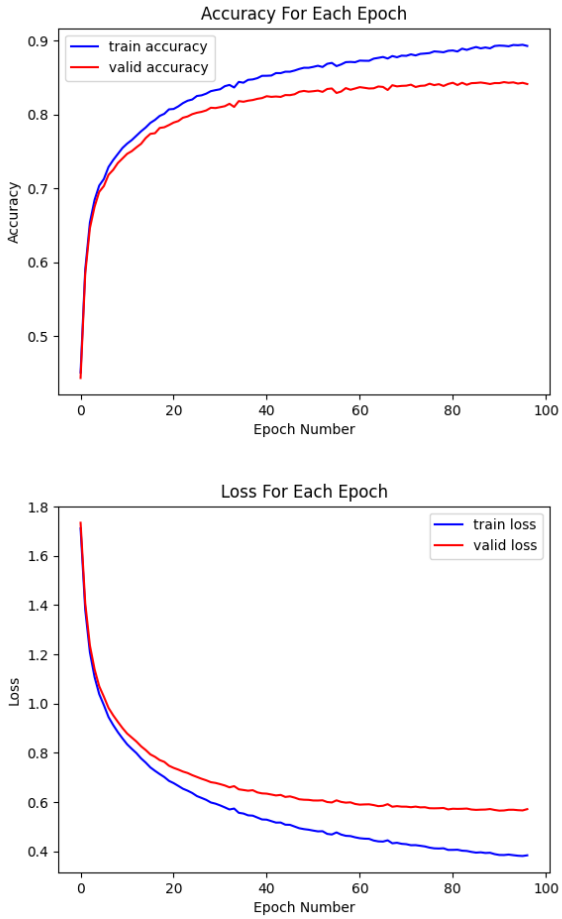

Figure 10: Activation = ReLU, Epoch = 150, Learning Rate = 0.003, Early Stop = 5, Momentum = 0.6, Final Accuracy = 83.4%

6 Network Topology

We changed the neural network structure by halving and doubling the number of hidden units. Changing to 64 from 128 hidden units resulted in an accuracy of 69.6% vs 128 to 256 resulting in accuracy of 73.8%. Increasing the number hidden units increased the accuracy because it allows more versatility in our model which is necessary when there are a decent amount of categories. Half of the original hidden units may have caused under-fitting, especially in a neural network with 1 hidden layer. We then tried to double the number of hidden layers to 2 with approximately 115 hidden nodes to equal the same amount of parameters as a single hidden layer of 128 which resulted in an accuracy of 79.3%. Having an additional layer also seems to have slightly less divergence between the training and validation which results in less over-fitting. The additional hidden layer with less hidden nodes may help with generalization.

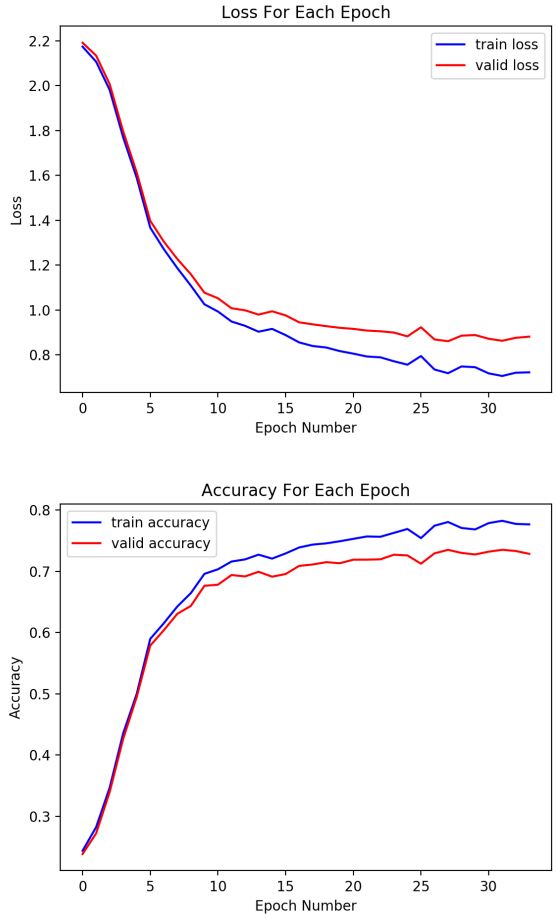

Figure 11: Hidden Units = 64, Activation = tanh, Epoch = 100, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 69.6%

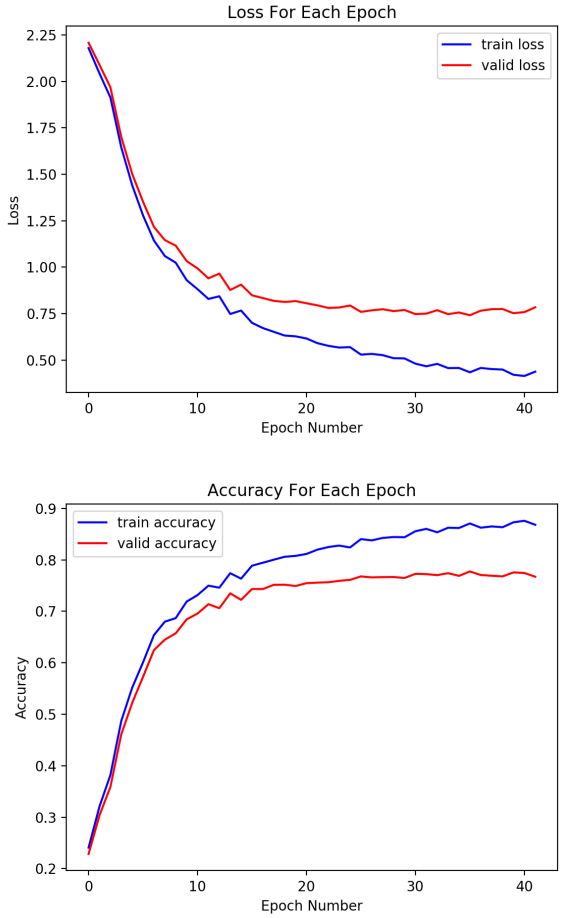

Figure 12: Hidden Units = 256, Activation = tanh, Epoch = 100, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 73.8%

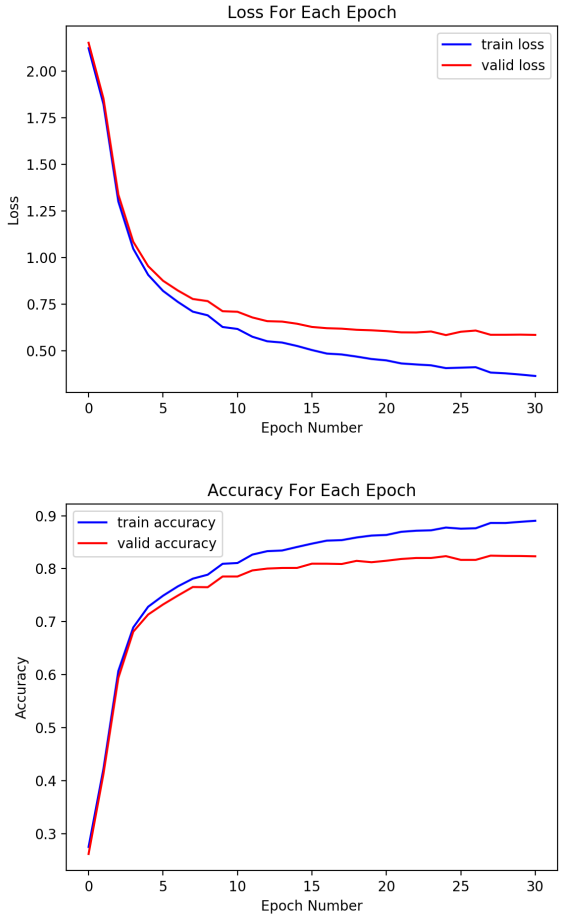

Figure 13: Hidden Layers = 2, Activation = tanh, Epoch = 100, Learning Rate = 0.005, Early Stop = 5, Momentum = 0.9, Final Accuracy = 80.6%

7 Team Contributions

Mitchell: Worked on implementing the neural network with Moses. Worked on forward pass, backpropagation, momentum, activation/layer classes, numerical approximation, and general training functionality. Helped with the writeup, including dataset, backpropagation, momentum, and activation function portions.

Moses (Jun Hwee): Worked on forward and backward pass in the neural network class. Implemented L1 L2 regularization. Worked on loss function as well as gradient descent. Wrote part of the writeup including Regularization, Network Topology.

8 Related work

[1] Gary Cottrell (2020) Backprop pp.22-29 UCSD

[2] Gary Cottrell (2020) Improving Generalization Lecture 4 pp.2-4 UCSD

[3] Gary Cottrell (2020) Tricks of the Trade pp.6, 46, 55 UCSD