Multi-Layer Neural Networks Implementation using Numpy

Andrew Ghafari
Department of Computer Science
University of California San Diego
aghafari@ucsd.edu

Jay Jhaveri
Department of Computer Science
University of California San Diego
jjhaveri@ucsd.edu

Abstract

In this assignment, we were handed a classification task to perform on the CIFAR-100 dataset. Unlike the previous assignment, we did not implement logistic regression and softmax regression, but instead we used multi-layer neural networks with a softmax output activation function, just because we are dealing with a multi-class classification task. We will not be using the full 100 classes that the dataset offers, but we will only restrict ourselves to 20 target labels, which are the 20 super classes of the dataset. In the last part of the problem, we just experimented with 100 classes but this was not the focus of the assignment. We started out with a 2 layer network, ie, an input layer, a hidden layer and an output layer, with the hidden layer having a tanh activation function and the output layer having a softmax activation function. Then throughout the problem improved the accuracy by changing the architecture of the model. We started by adding momentum, then regularization, then we moved on to experimenting with different hidden activation functions, and finally increased the number of hidden layers of the networks. The last task, was to use our best network found throughout the process, and train it onto the 100 classes and check its performance. Let us now walk you through our results and interesting findings. In short, we started with normalizing the data using z normalization, which is popular when dealing with images as training data (we will explain it in depth later), and passed it through a neural network of 2 layers, tanh and softmax activation functions and a momentum of 0.9, we first got an accuracy percent score of 24.46%. We then started improving it, we added regularization term, which was another hyperparameter that we tuned, and set it finally to 0.01, this improved our accuracy to 24.74%. Next, we experimented with sigmoid and ReLU activation functions for hidden layers which got us 24.17% and 27.84% accuracy respectively. This can be explained by ReLU being simple and effective in not firing up all units simultaneously which leads to better learning and better convergence. Next, we changed the network topology, by first halving the number of hidden units, this reduced our accuracy to 25.73% , which was expected because there is less 'learning' happening. Then we doubled the original number of hidden units, and this bumped our accuracy to 28.21%, which was also expected. After that, we kept the number of parameters same, but increasing the number of layers of the overall network, which got us an accuracy of 27.97%. Finally, we projected our best network found throughout this whole process and tried it on the original 100 classes of the CIFAR-100 dataset, this reduced our accuracy to 17.55%.

1 Introduction

The CIFAR-100 is a benchmark dataset for computer vision and image classification tasks. It consists of 60,000 32x32 color training images and 10,000 test images divided into 20 super classes, and 100 classes overall. The dataset is widely used for evaluating the performance of computer vision algorithms, particularly deep learning models. In this programming assignment, we were tasked to perform classification on this dataset, using a multi-layer neural network architecture developed from scratch without outsourcing functionalities from libraries, and only using numpy. The individual part covered the mathematical proofs behind forward and backward propagations which we found crucial when we were doing the programming part, especially the vectorization part, since all of the conceptual problems that we would have faced while coding the network here were already solved there. We first started with a vanilla implementation of a 2 layer NN, with the hidden layer having 128 neurons and a tanh activation function, and the output layer having 20 neurons and a softmax activation layer. We used minibatch gradient descent to minimize the multi-class cross entropy loss function with a starting batch size of 1024. The weights in these layers were updated using the method called Backpropagation. This is done so that the output produced is as close as possible to the desired output. The algorithm calculates the error between the desired output and the actual output, and then adjusts the weights in the network to minimize that error using SGD. This process is repeated until the error reaches an acceptable level or a maximum number of iterations is reached. In this version of backpropagation we added a patience term, which is basically a counter of how many times our validation loss increased consecutively. We set it to 5, which means that after 5 consecutive epochs of validation loss increasing the epoch iteration will break because this constant increase in loss indicates that the model started to overfit on the training data and we just early stop the model.

After that, we improved the model's accuracy by adding the momentum term. Momentum, as explained in class, is a heuristic aimed at improving the stability and speed of gradient descent in deep learning in general. It helps prevents oscillation in the updates, by reducing the learning rate whenever the updates terms are oscillating (indicating we are missing the minimum) and increasing it if we are going in the right direction. This got us around 25% accuracy for the test dataset which is relatively close to the 29% benchmark given. We also did hyperparameter tuning by manually implementing grid search using 3 sets of for loops, to optimize numerous variables which are lambda, the regularization term that we will discuss next, batch-size, and learning rate, and every score mentioned here is utilizing the hyperparameters found which were 0.01 for lambda, 1024 for batch size and 0.005 for learning rates.

After that, we added regularization, more specifically L2 regularization. L2 regularization is usually called Ridge regularization, the main goal behind it is to prevent our model from overfitting on our training data and failing to generalize for unseen data. It does that by adding a penalty term to the loss function, penalizing by the square of the weight coefficients, encouraging small ones and overall reducing the complexity of the model. It is in simple terms a tradeoff factor between model accuracy and model complexity. Adding this term improved our accuracy by a little bit, from 24.71 to 24.74%.
Furthermore, we experimented with different activation functions. We did that by using the optimal network found before and trying different activations functions for the hidden layer, so substituting tanh, with ReLU and Sigmoid. For sigmoid, it decreased our overall test accuracy by almost 0.6%, but on the other hand for ReLU we saw a bump in accuracy reaching 27.84%. Moreover, we tried 3 more variations of the network topology, the first being halving the overall numbers of hidden units, which obviously decreased our accuracy to 25.73%, doubling it, which bumped it to 28.21% and doubling the number of layers while keeping the number of parameters same, which got us a 27.97%. The last thing we did is just an extrapolation, we just tried our best model found throughout the whole programming

assignment and tried it on the 100 classes. This definitely reduced the overall accuracy, since now we have 100 classes without additional training data, and using the same “shallow” considered network for a much bigger task. We got an accuracy rate of 17.55% using the best hyperparameters and the optimal network architecture found previously.

In the following sections, we will start off by talking about related work, or papers that inspired us in general. Next, we will be discussing pre-processing done on the images in the dataset and how it became ML-ready. Following this, we will be showing the results of implementing the algorithms implemented from PCA to logistic Regression to softmax regression. Finally, we will end this paper by providing team members' contributions and the source code.

2 Related work

We first would like to thank Prof Cottrell for the slides posted as we heavily relied on them while doing this programming assignment. More specifically, Lecture 3, 4 and 4A from the Lecture Notes Winter 2023 on Piazza. In addition to that, we would also like the thank the TAs for the help they provided whether on piazza or in person during office hours.

We would also like to mention some of the articles and blogs that we read to get a better understanding of some of the concepts: Brownlee (2021) helped us in understanding the implementation of momentum in gradient descent in general [1], concepts wise. Bushaev (2018) talks about about implementing SGD with momentum [2], and Tewari (2022) is about implementing regularization as well [3].

Lastly, Sichkar (2023), although a more advanced implementation of neural network using covnets, helped us get a general idea about the CIFAR dataset itself [4].

3 Dataset

3.1 Description and Loading

The dataset provided to us is CIFAR-100, consisting of 60,000 32x32 color training images and 10,000 test images. The images are divided into 100 fine grained classes and 20 super classes.

To visualize the dataset, we randomly selected an image of each of the 20 classes in the dataset.

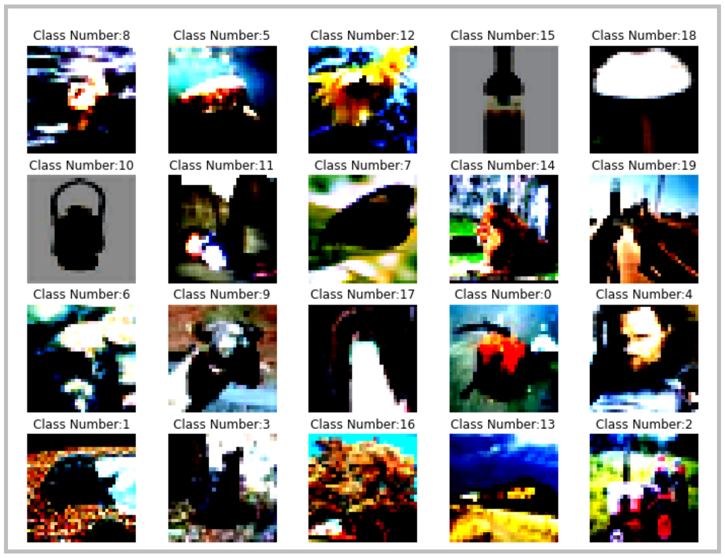


Figure 1: Random Depiction of Each Classes in the Dataset

These are some descriptive analysis we performed on the dataset.

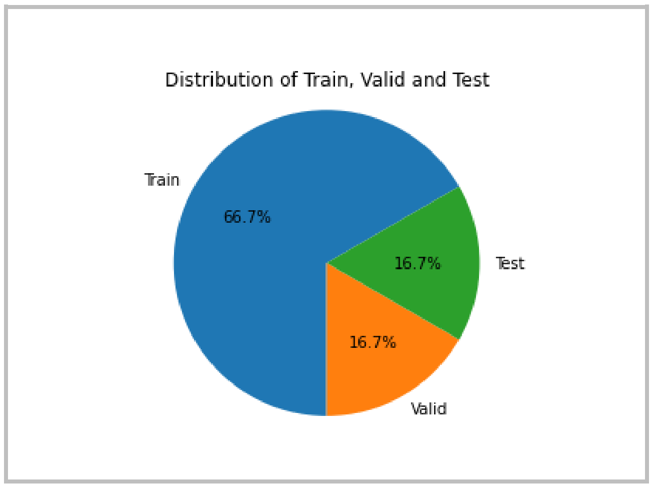


Figure 2: Percentage partition between train and test

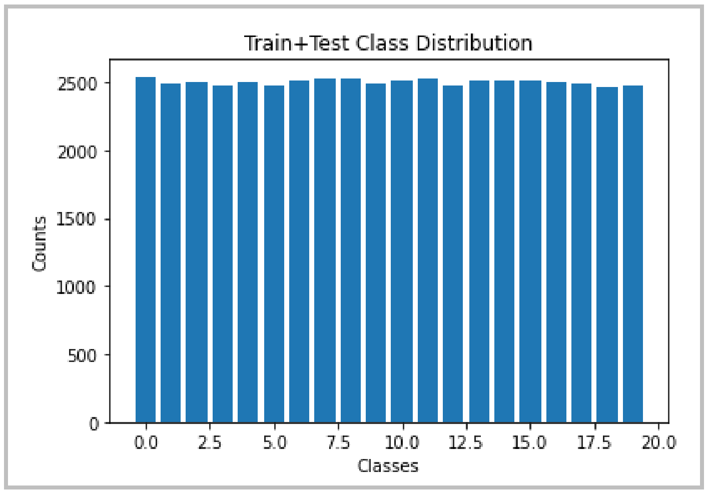


Figure 3: Total dataset partition between classes

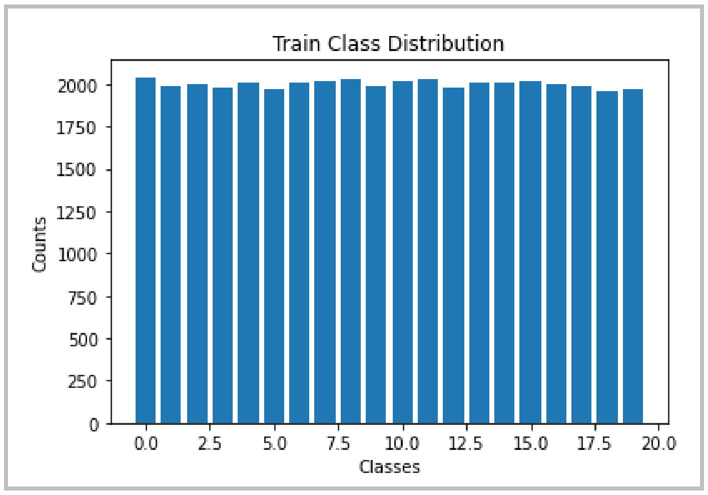


Figure 4: Train dataset partition between classes

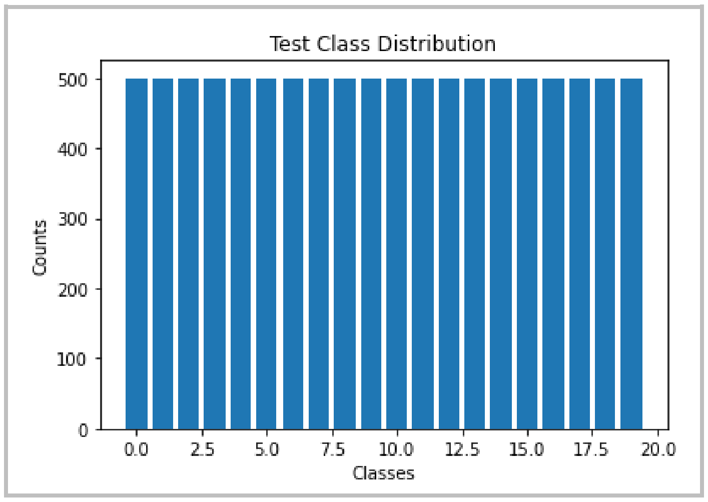
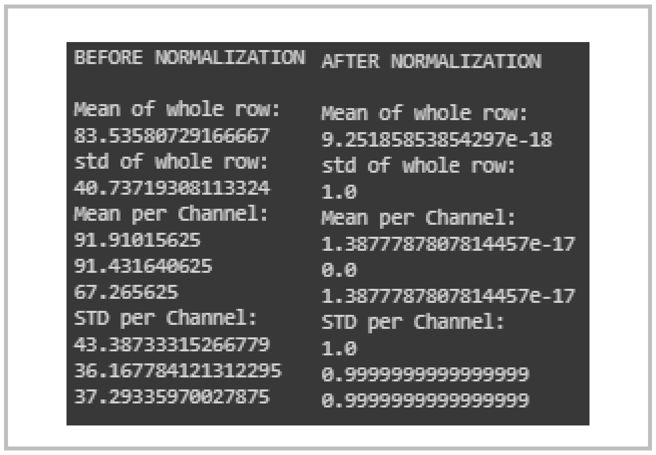


Figure 5: Test dataset partition between classes

As we can see, the data is fairly partitioned between train and test, and it is also uniformly spread across classes, which contributed to the model's success.

The data was normalized using z-score normalization, which is effectively a method of rescaling the values of the pixels of the images to have zero mean and a unit standard deviation. In this dataset, the figures have 3 channels, each channel corresponding to a primary color (R, G, B). Here we did the normalization per channel. As described in class, this is a good idea because by doing so, we are ensuring that each channel of each image has a mean of 0 and a standard deviation of 1. This will stabilize the networks to deviations in scale between the different channels. This is specifically important when dealing with images, because it helps train the deep learning model better and debias it from changes in intensities of colors.

Here are the mean and std statistics of a single image as a whole and per channel before and after doing z score normalization



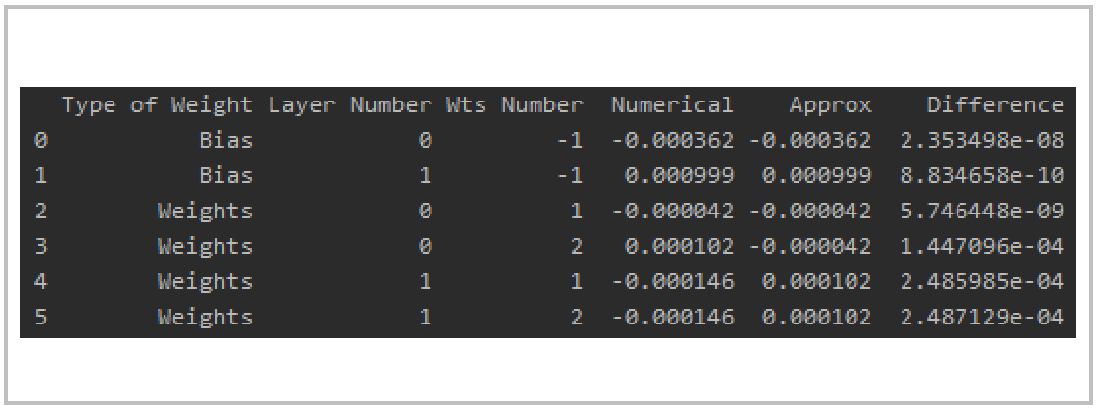
```
BEFORE NORMALIZATION AFTER NORMALIZATION

Mean of whole row:    Mean of whole row:
83.53580729166667     9.25185853854297e-18
std of whole row:     std of whole row:
40.73719308113324     1.0
Mean per Channel:     Mean per Channel:
91.91015625           1.3877787807814457e-17
91.431640625          0.0
67.265625             1.3877787807814457e-17
STD per Channel:      STD per Channel:
43.38733315266779     1.0
36.167784121312295    0.9999999999999999
37.29335970027875     0.9999999999999999
```

Figure 6: Statistics before and after

4 Numerical Approximation of Gradients

In this question, we did a sanity check for our back propagation formula. We did so by comparing the gradient computed using the numerical approximation formula provided for us, and the one calculated during our backpropagation function. Below is the table requested that shows, type of weight, gradient obtained from numerical approximation, gradient obtained by backpropagation, absolute difference between the two.

Please keep in mind that epsilon used is 10^{-2}, so differences should be in the order of 10^{-4} which is what we are getting



```
  Type of Weight Layer Number Wts Number  Numerical    Approx    Difference
0           Bias            0         -1  -0.000362 -0.000362  2.353498e-08
1           Bias            1         -1   0.000999  0.000999  8.834658e-10
2        Weights            0          1  -0.000042 -0.000042  5.746448e-09
3        Weights            0          2   0.000102 -0.000042  1.447096e-04
4        Weights            1          1  -0.000146  0.000102  2.485985e-04
5        Weights            1          2  -0.000146  0.000102  2.487129e-04
```

Figure 7: Sanity Check Table for Backpropagation

5 Neural Network with Momentum

We implemented our neural network architecture as follows: we had a 2-layer network (with inputs' layer it becomes a 3 Layer Network). The input layer size is 3072 (+1 Bias), the hidden layer has a size of 128 (+1 Bias) with a tanh activation function, and the output layer has 20 units to represent the 20 super classes that we are classifying on, with a softmax activation function to be able to perform multi-class classification. We also included a momentum of 0.9, as requested, which essentially help us converge faster and better, by making our step bigger if we are going in the right direction and slowing us down if we are going in the wrong one. More of these details about the effect on momentum on gradient descent have been explained above.

The hyperparameters that we did tuning on were learning rate, batch size using for loops, and we found out that the best learning rate was 0.005 and best batch size 1024. Below are the plots for train/val loss and accuracy, along with the test accuracy reported.

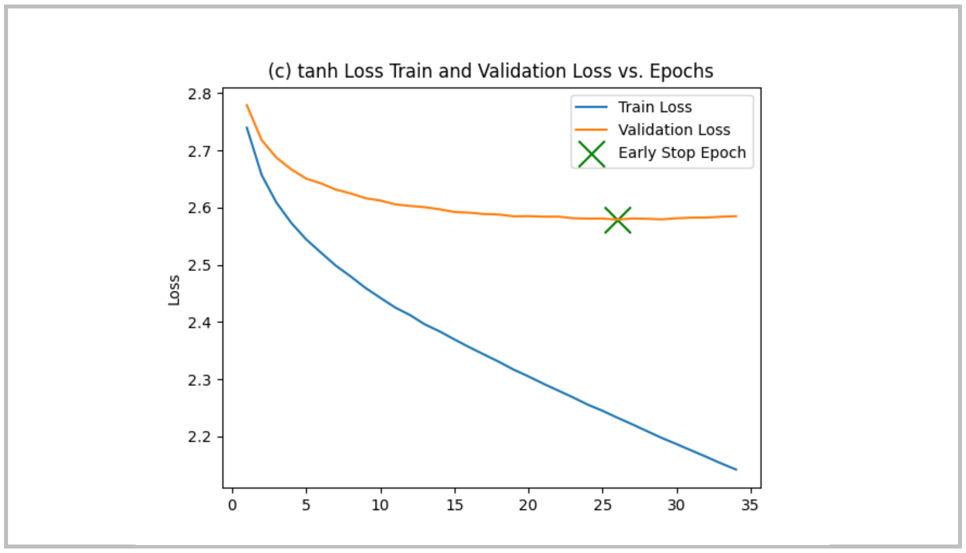


Figure 8: Momentum Loss plots [train and valid]

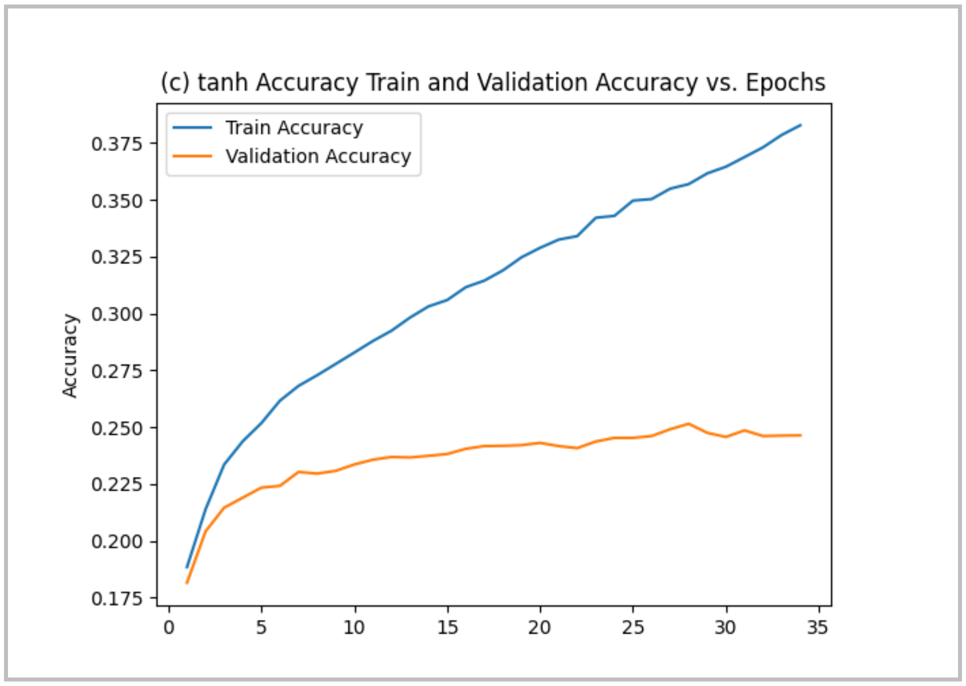


Figure 9: Momentum Accuracy plots [train and valid]

This network performed well, as we got 24.46% as our test accuracy on unseen data, so it also generalizes well, which is a good sign. Now the accuracy isn't in the high 80s because it is a relatively shallow network to be able to predict 20 classes, and we don't have enough data to offset this effect. But overall the plots seem okay, there is convergence happening for the loss function and we got a close to desired results.

6 Regularization Experiments

In this section, we introduced a new hyperparameter, which is the regularization term λ for L2 regularization. We started with the best network found and discussed previously. We added λ in our update parameters functions using the technique called weight decay, where we subtract a small multiple factors parameterized by λ from the weight update to try and prevent weights from becoming more and more dominant in the model.

We tuned λ between values of 0.01, 0.001, 0.0001 and got the best result when it was equal to 0.01. Although we did not see a huge bump in accuracy, it only increased it by just a bit. Here are the new plots.

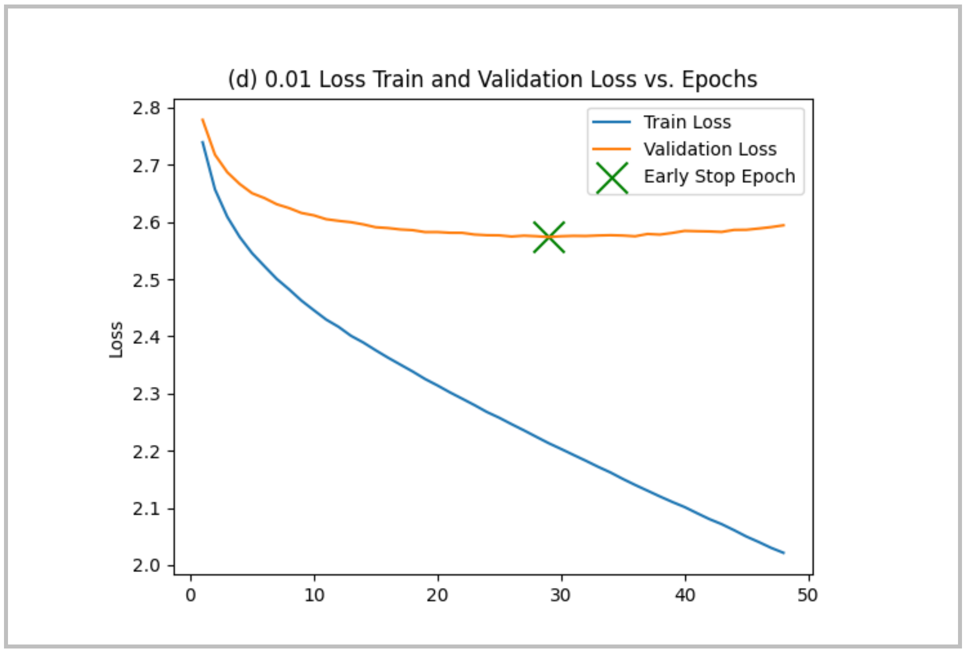


Figure 10: Loss plots [train and valid] with λ

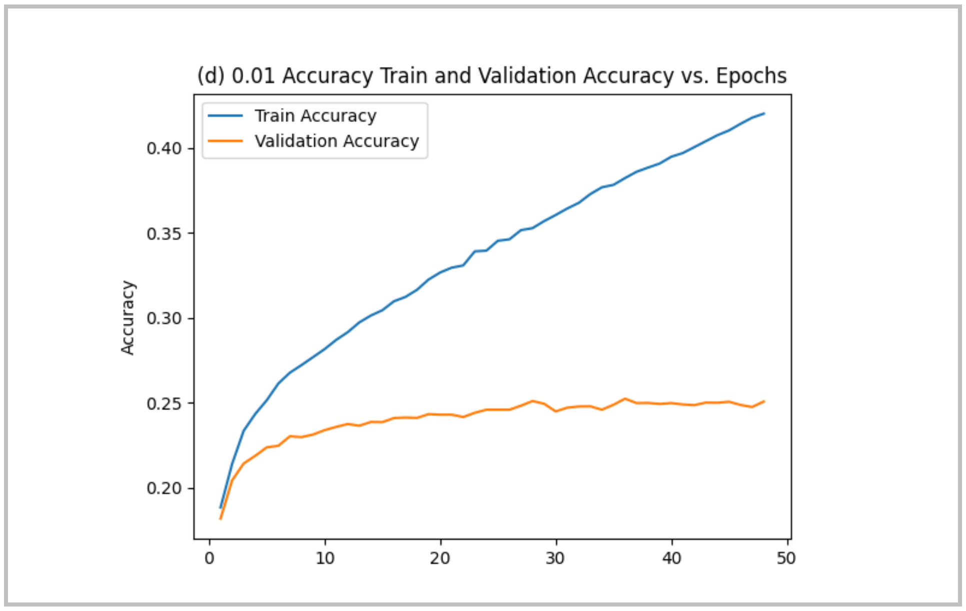
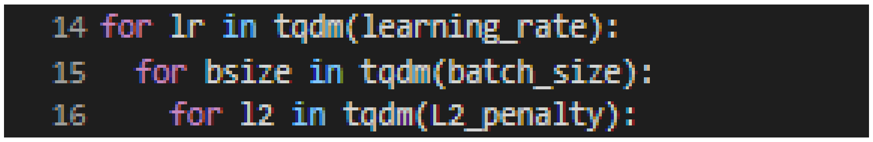


Figure 11: Accuracy plots [train and valid] with λ

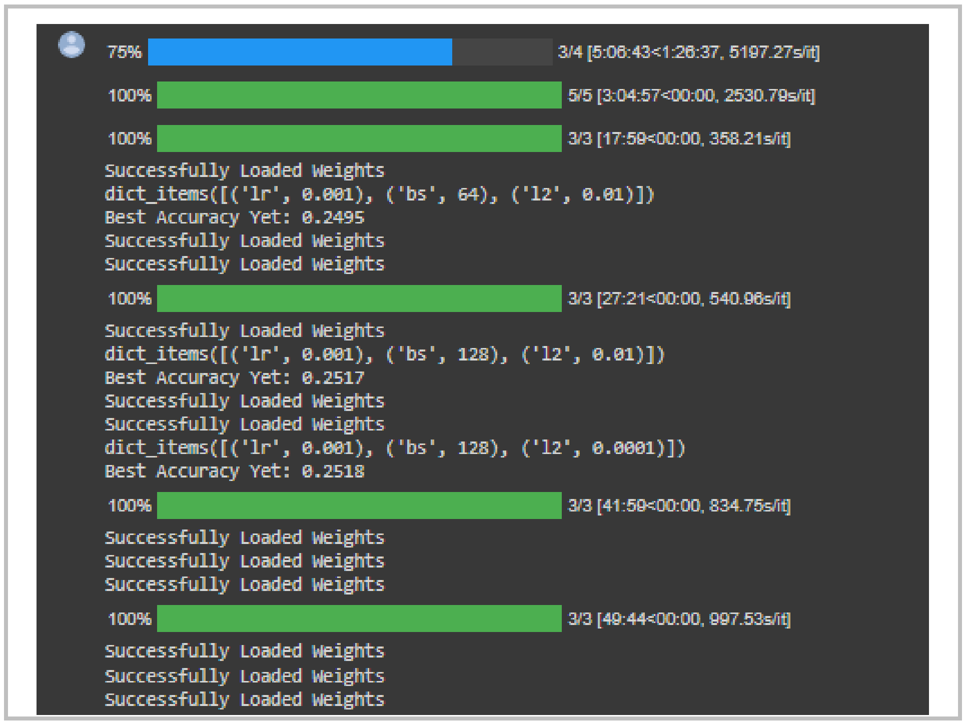
As I said, this increased our overall test accuracy from 24.46 to 24.74 which is objectively a small jump. I think this can be explained by two things: our model might not have been overfitting before so adding this parameter wont help it do much, and I get to say that because our model did not memorize train data and could not generalize to test data, but in fact the validation accuracy and test accuracy were actually pretty close to each other. And L2 usually penalizes big weights more which we did not quite have, and this might be the reason we didn't see a huge jump in accuracy after adding the regularization term.

We are adding a snapshot of the code used to perform hyperparameter tuning and its result mid way through the search. We performed the technique called Grid Search manually by the use of for loops:



```
for lr in tqdm(learning_rate):
  for bsize in tqdm(batch_size):
    for l2 in tqdm(L2_penalty):
```

Figure 12: Grid Search Source Code Snapshot



```
75% 3/4 [5:06:43<1:26:37, 5197.27s/it]
100% 5/5 [3:04:57<00:00, 2530.79s/it]
100% 3/3 [17:59<00:00, 358.21s/it]
Successfully Loaded Weights
dict_items([('lr', 0.001), ('bs', 64), ('l2', 0.01)])
Best Accuracy Yet: 0.2495
Successfully Loaded Weights
Successfully Loaded Weights
100% 3/3 [27:21<00:00, 540.96s/it]
Successfully Loaded Weights
dict_items([('lr', 0.001), ('bs', 128), ('l2', 0.01)])
Best Accuracy Yet: 0.2517
Successfully Loaded Weights
Successfully Loaded Weights
dict_items([('lr', 0.001), ('bs', 128), ('l2', 0.0001)])
Best Accuracy Yet: 0.2518
100% 3/3 [41:59<00:00, 834.75s/it]
Successfully Loaded Weights
Successfully Loaded Weights
Successfully Loaded Weights
100% 3/3 [49:44<00:00, 997.53s/it]
Successfully Loaded Weights
Successfully Loaded Weights
Successfully Loaded Weights
```

Figure 13: Results mid-way through the grid search

7 Activation Experiments

In this section, we experimented with the remaining of the activation functions that we had defined up, so we replace the tanh activation function of the hidden layer by ReLU and sigmoid activations respectively. As explained, this doesn't change the structure of the code, it just changed the derivative calculation for the hidden units while backpropagating through the network to perform weight updates. We will now show the results we got first for sigmoid, and then for ReLU and discuss results.

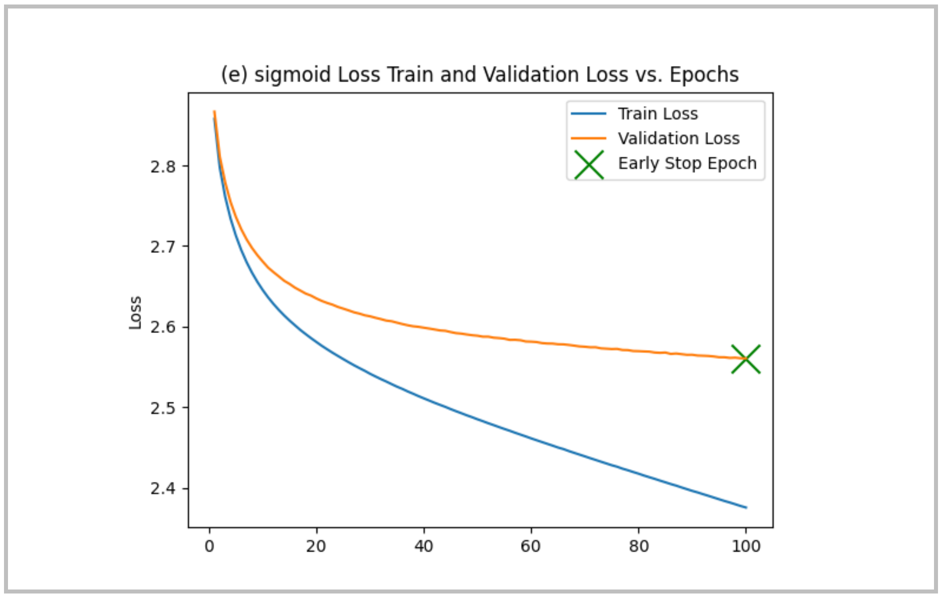


Figure 14: Sigmoid loss plots [train and valid]

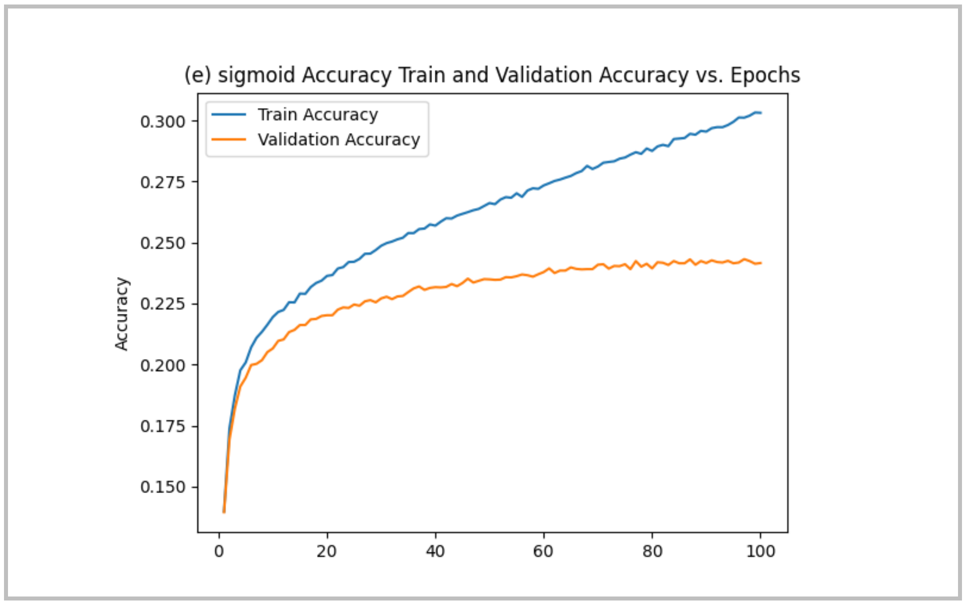


Figure 15: Sigmoid Accuracy plots [train and valid]

We achieved a test accuracy of 24.17% which was less than what we had before, and this was kind of expected since sigmoid is not a natural choice for hidden layer and definitely not for a multi-class classification task. This is due to the fact that the sigmoid activation function saturates at high positive and high negative values which can lead to vanishing gradient problems and affects the model's overall accuracy. Now we will move to discussing ReLU instead. This time, the new activation function gave impressive results, with a bump in accuracy to 27.84%. ReLU performed better which was also expected, and it is one of the reasons why it is so popular as a hidden layer activation function. It not only speeds us the process by having a very easy to calculate derivative, but it also alleviates the vanishing gradient problem due to the nature of it, and this is primarily the reason we saw a bump in accuracy. These are the corresponding plots for this part.

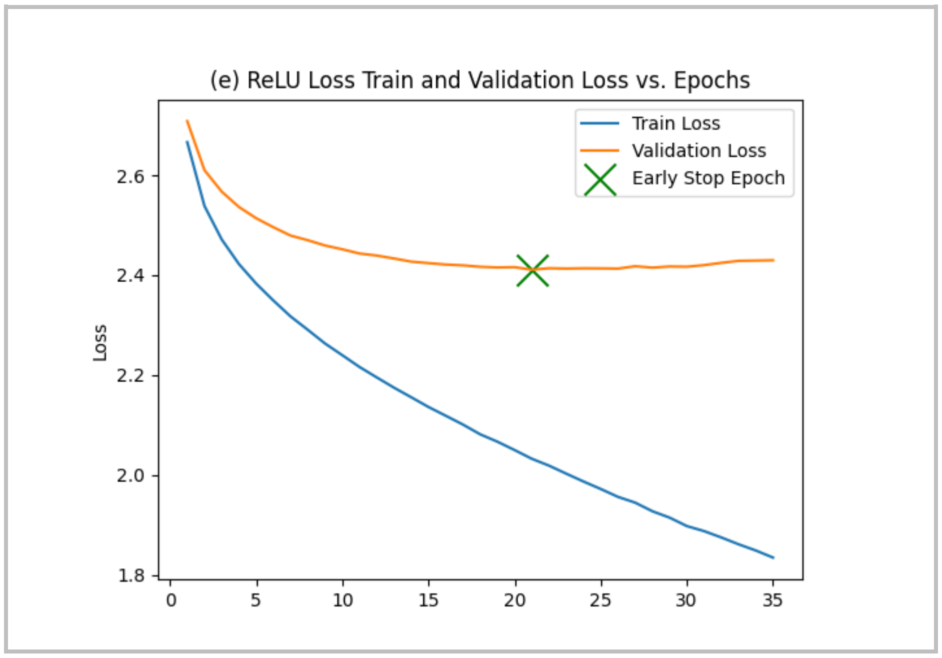


Figure 16: ReLU loss plots [train and valid]

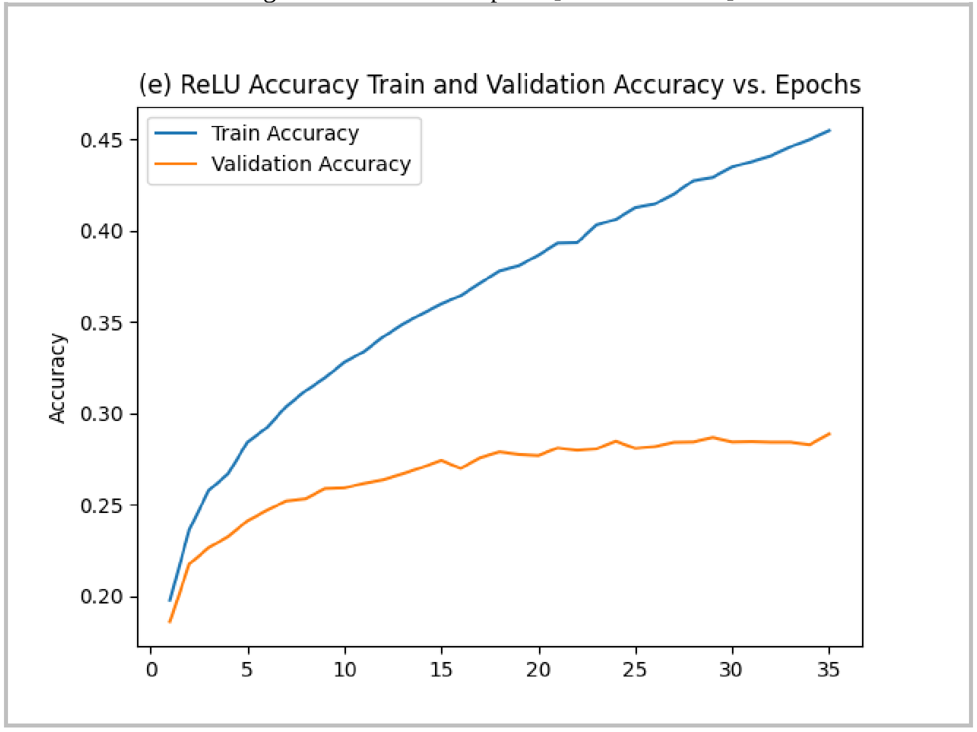


Figure 17: Sigmoid Accuracy plots [train and valid]

As we previously discussed, ReLU gave better results than sigmoid and even better than tanh previously used, and therefore we will be using it for the upcoming parts.

8 Experiment with Network Topology

In this section, we experimented with the network topology using the best network found until now (till last section). There are 3 tweaks that we performed, we will discuss them and their results.

We first tried halving the number of hidden units, ie decrease it from 128 to 64, which reduces the number of parameters of the model and overall limits the ‘learning’ even more, which was visible by the reduction in accuracy metric that we got. Our test accuracy dropped to 25.73% which as I said before was kind of expected. These are the corresponding plots.

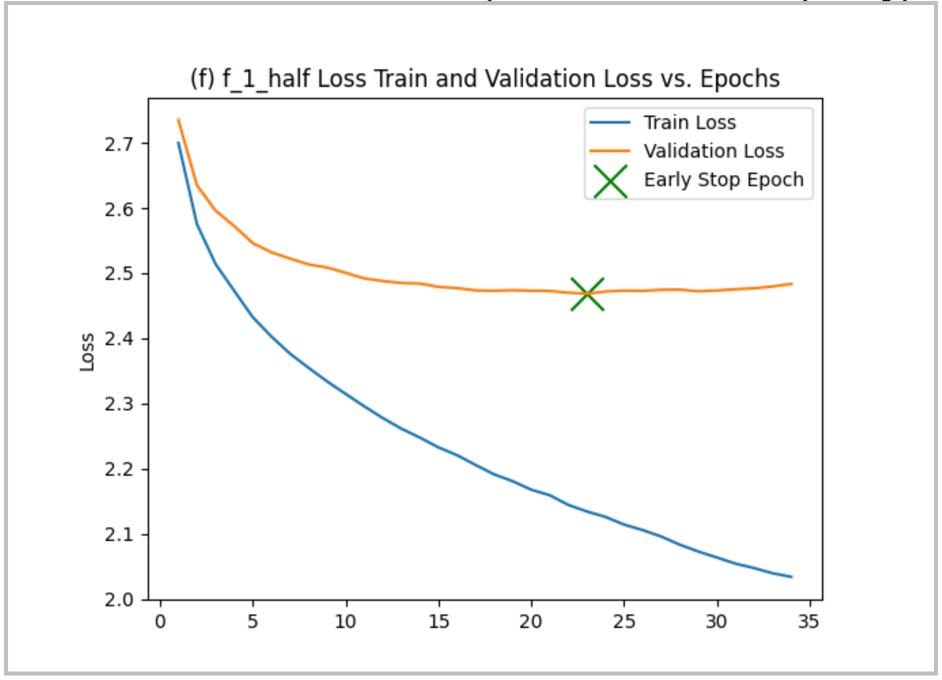


Figure 18: Halving hidden units loss plots [train and valid]

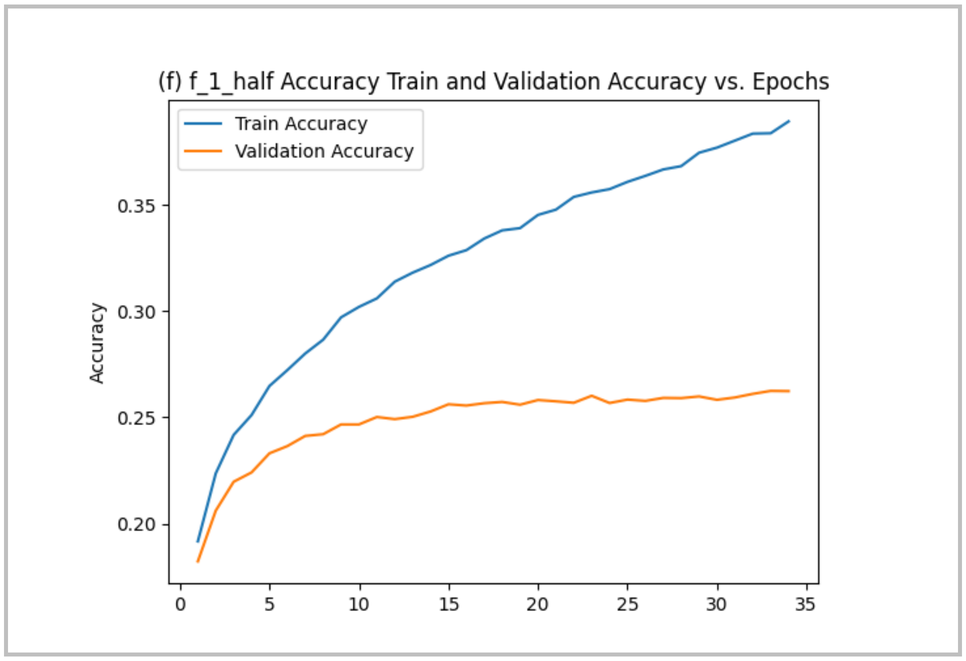


Figure 19: Halving hidden units Accuracy plots [train and valid]

Then we tried the opposite, which is increasing the number of hidden units from 128 to 256 in the hidden layer, and that imposes having more parameters and weights to train which overall improves the results of the network on this classification task. This was also reflected in the results and the plots we got. Our test accuracy for this network came up to 28.21% which was an impressive jump.

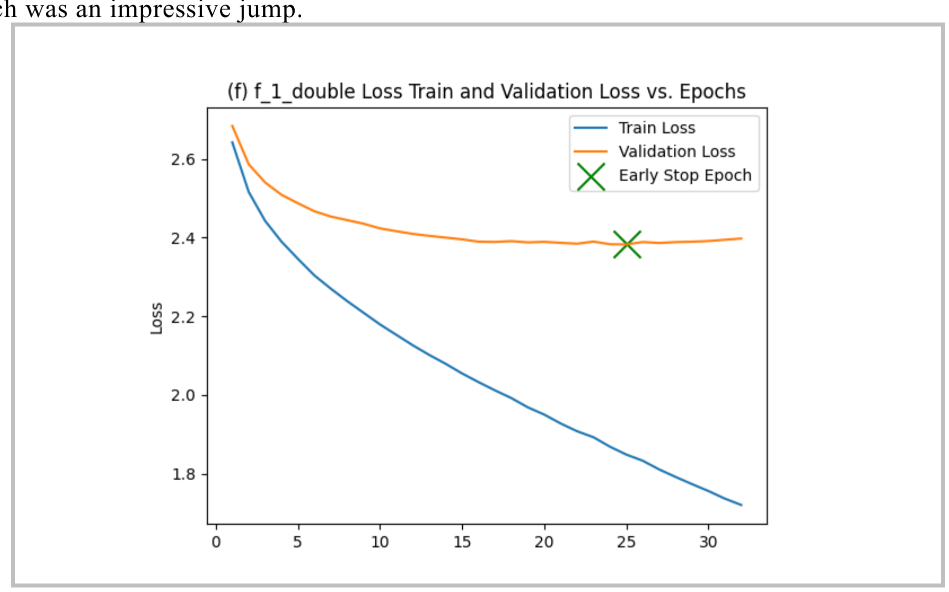


Figure 20: Doubling hidden units loss plots [train and valid]

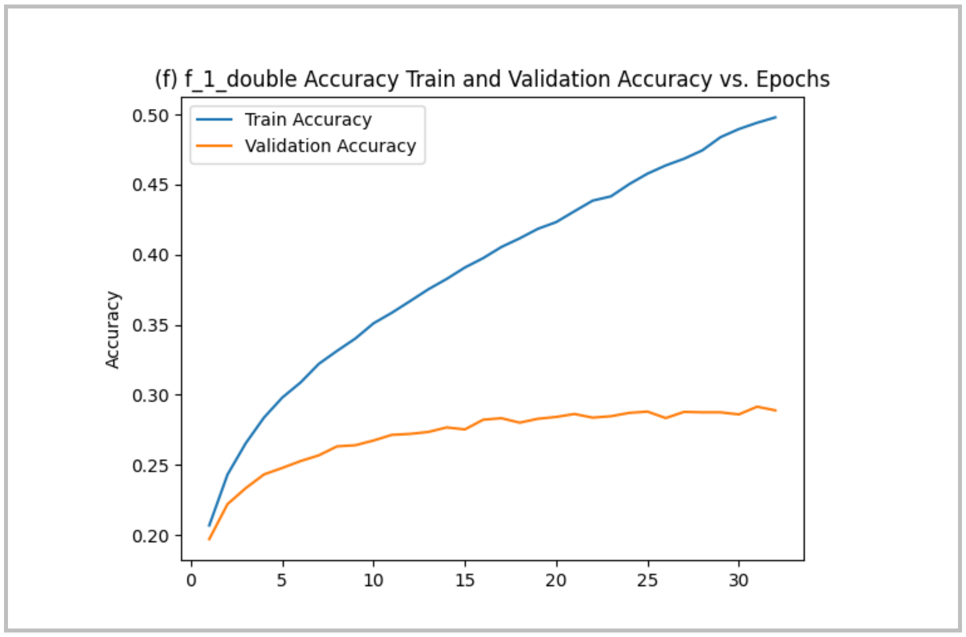


Figure 21: Doubling hidden units Accuracy plots [train and valid]

The last thing we did with regards to network topology is actually increasing the number of layers rather than modifying the number of parameters, so instead of having 2 layers, we increased that number to 3 without increasing the number of parameters, so now our architecture is the following: 3072 input units, 123 hidden units for hidden layer 1, 123 hidden units for hidden layer 2 and an 20 output units for the output layer. This leaves us with 3072*123+123*123+123*20 which is almost equivalent to the number of params we had previously which was

3072*128+128*20. So in this way, we helped conserve the number of params but with a different network architecture. This also improved our results from part e, even though it wasn't as good as when we doubled the total number of hidden units (there we had more params) but it was still better than then previous [3072,128,20] architecture that we previously had. We achieved a test accuracy of 27.97% for test data. This can be explained by the fact that having more layers will allow a more structured representation of the image data that we have which usually results in a better generalization performance and a better modelling of complex representations. Following are the loss and accuracy plots.

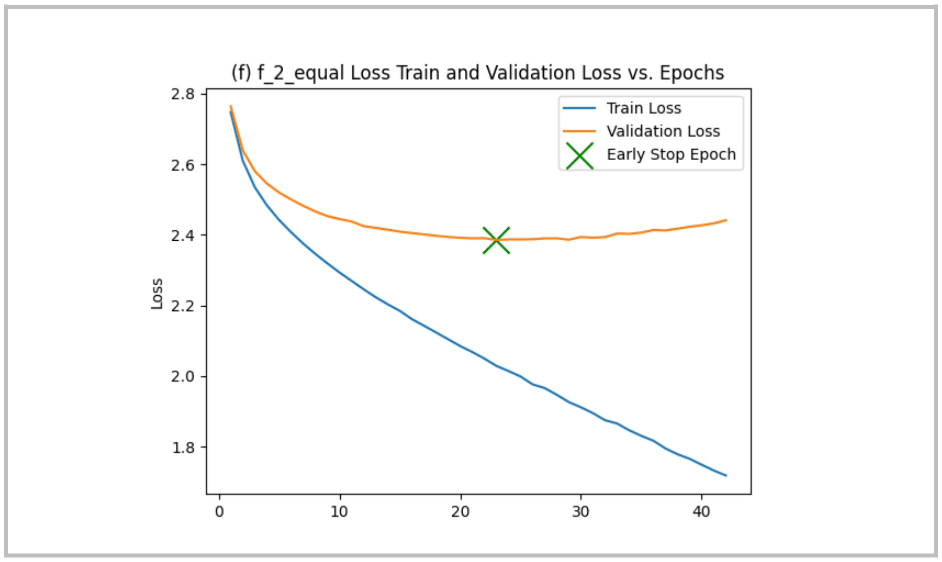


Figure 22: Adding one more hidden layer loss plots [train and valid]

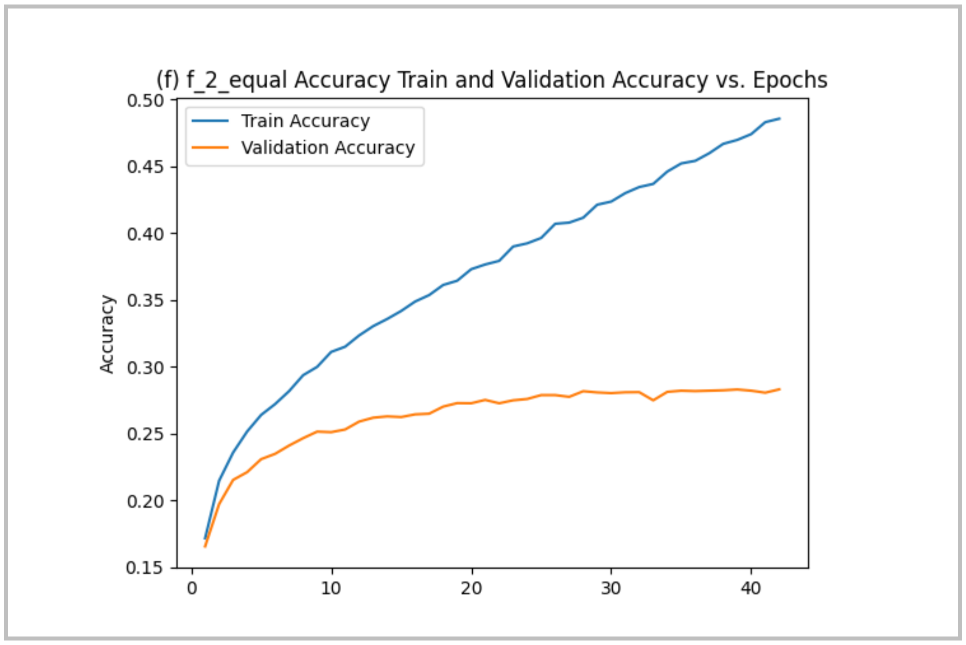


Figure 23: Adding one more hidden layer Accuracy plots [train and valid]

9. Experiment with 100 classes

In this section, we are using the best architecture we found up to part (e) on 100 classes of the dataset instead of 20 coarse ones we were previously using. To reiterate, the best accuracy we got after hyperparameter tuning by grid search is momentum Gamma = 0.9, regularization lambda = 0.01, batch size = 1024, and learning rate = 0.005. Following are the plots we got for the train and validation accuracy and loss over the epochs.

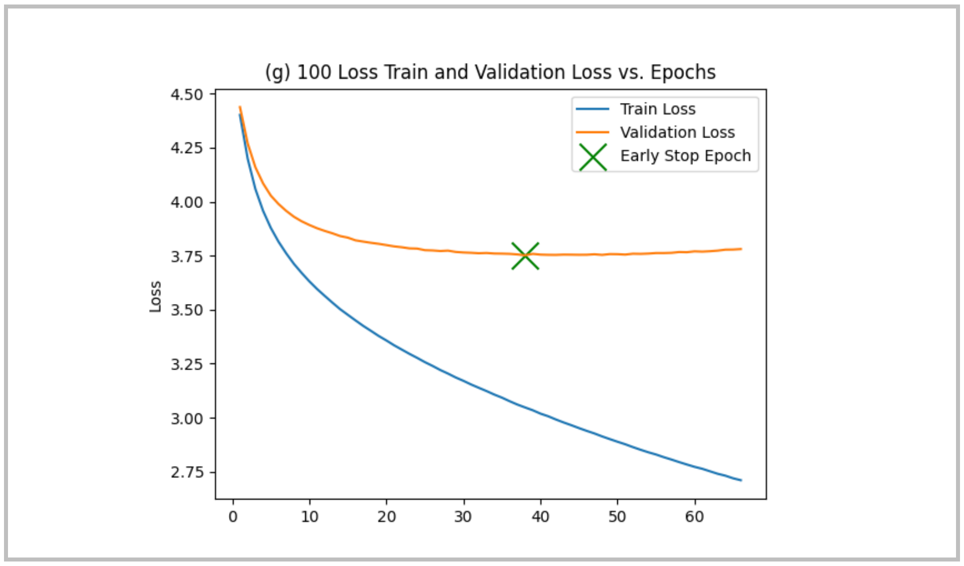


Figure 24: 100 Classes Loss plots [train and valid]

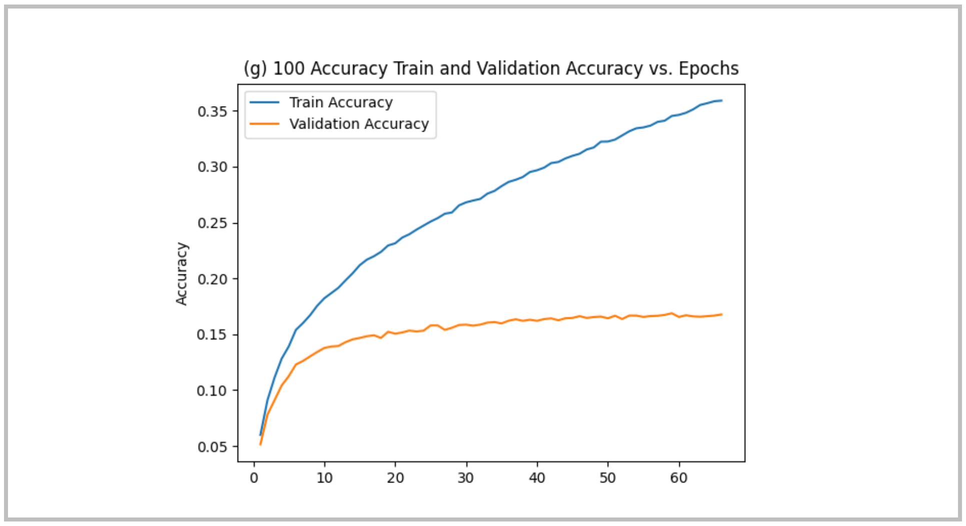


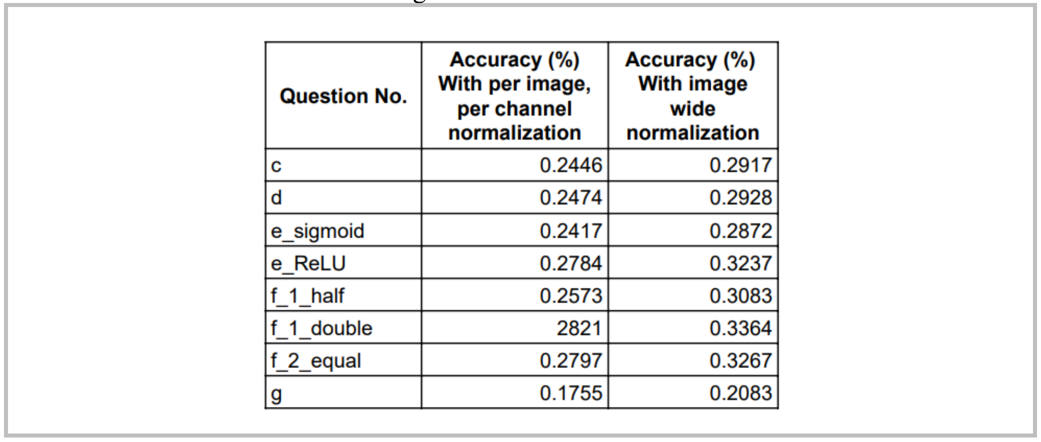
Figure 25: 100 Classes Accuracy plots [train and valid]

We achieved a test accuracy of 17.55%. This is higher than what we were expecting. In our opinion the performance drops as compared to the results before, because we are using the same “Shallow” network with even less training examples per class without increasing the number of parameters available to train. To improve this, we suggest increasing the number of hidden units while also increasing the number of hidden layers. Further, as taught in class, we can introduce convolution filters as part of our training network, as convolution combined with techniques such as pooling is much better in detecting patters and becoming invariant to most of the noise seen in images. Lastly, more data per classes can never hurt!

10 Results and Observation:

The most fascinating thing we found while tuning the hyperparameters is that changing the way we normalize the images matters a lot! The problem statement says that because the given images are 3-channels, specifically R,G, and B and hence we should normalize them via the channels, we remembered professor mentioning normalizing throughout the image at once also works, and we tried that. Although, logically and theoretically, normalizing channel-wise should give better results because it makes the model invariant to change in intensities of individual color channels, in this dataset, normalizing the whole image at once gave a ~5% boost in accuracy on average.

We quickly posted about this discovery on Piazza, and were instructed to write these observations in our report and report the results with and without channel wise normalizing. The results are shown in the following table:



Question No.	Accuracy (%) With per image, per channel normalization	Accuracy (%) With image wide normalization
c	0.2446	0.2917
d	0.2474	0.2928
e_sigmoid	0.2417	0.2872
e_ReLU	0.2784	0.3237
f_1_half	0.2573	0.3083
f_1_double	2821	0.3364
f_2_equal	0.2797	0.3267
g	0.1755	0.2083

Figure 26: Overall Results per question, per normalization.

11 Team contribution

We divided the project equally in all aspects. In fact, we are roommates so we did the whole project part together. For programming, it was pair programming with both of us on discord one guy sharing his screen and coding on a shared Google Colab notebook, alternating this role between different sections of this assignment.

Andrew normalized the dataset per image per channel (3-a), split the data into train validation and testing. In addition to that, he implemented backward propagation function and checked it by doing part 3-b. He also did part e, which is the experimenting with different activations.

Jay implemented the activation functions formulas, their derivatives, and the forward propagation function. In addition to that, he implemented momentum and regularization problems. Plus, he also did the experimenting with network topology and 100 classes.

For the project report, it was pretty straightforward to do, since we had done everything together, we split up the writing in half, each one of us took the lead in writing the sections that he took the lead in coding.

12 Source code

The readme attached in source code pretty much explains how to run it.

References

[1]. Brownlee, J. (2021, October 11). Gradient descent with Nesterov momentum from scratch. MachineLearningMastery.com. Retrieved February 6, 2023, from https://machinelearningmastery.com/gradient-descent-with-nesterov-momentum-from-scratch/

[2]. Bushaev, V. (2018, June 21). Stochastic Gradient Descent with momentum - Towards Data Science. Medium. https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

[3]. Tewari, U. (2022, January 4). Regularization — Understanding L1 and L2 regularization for Deep Learning. Medium. https://medium.com/analytics-vidhya/regularization-understanding-l1-and-l2-regularization-for-deep-learning-a7b9e4a409bf

[4]. CIFAR-10 image classification with numpy only. cifar10. (n.d.). Retrieved February 6, 2023, from https://sichkar-valentyn.github.io/cifar10/#functions-for-dealing-with-cnn-layers