# A Manual Approach to Multi-layered Neural Networks

**Yigit Korkmaz A59010764,**
ykorkmaz@ucsd.edu

**Marcus Schaller A59004764**
mschalle@ucsd.edu

## Abstract

Neural networks are are a common tool in analyzing and classifying data. This paper will discuss the use of multi-layered network made without the help of high level machine learning software. When tested on the CIFAR-10 dataset we achieved a maximum accuracy of 50.24 percent.

## 1 Introduction

This experiment involves creating a multilayered neural network without using a high level machine learning library such as PyTorch. It was tested on the CIFAR-10 dataset which contains 10 classes and 50000 32x32 pixel RGB images. Throughout this paper our findings after experimenting with changing the hyper-parameters and architecture of the network will be discussed.

## 2 Related Work

Traditionally many other researchers use convolutional neural networks to classify the CIFAR-10 dataset. These often have a much higher accuracy than the fully connected network discussed in this paper. One such paper uses a recurrent CNN that was able to obtain an accuracy of 97.47 percent [2]. This attempt uses parameter sharing and introduces a novel ”soft sharing scheme” which resulted in substantial computational savings.

## 3 Methods and Results

### 3.1 Gradient error methods

In order to correctly implement a neural network, it is important that we first implement back propagation. In order to do this we used 10 data examples, one from each category, and run backpropagation to calculate the gradient. We then use the following equation in conjunction with the fact that $\epsilon = 10^{-2}$ to approximate the gradient.

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

## 3.2 Gradient error results

The goal is for the difference between the gradient calculated through backpropagation and the numerical approximation is to be within $10^{-4}$, since our $\epsilon = 10^{-2}$. We achieved the following values. Clearly we are within our desired goal and our back-propagation is effective.

Table 1: Calculated Gradient, Approximated Gradient and their difference

| Layer | Approximated | Calculated | Difference |
|---|---|---|---|
| Input to Hidden 1 weight 1 | -8.49E-11 | -4.14E-12 | 8.07E-11 |
| Input to Hidden 1 weight 2 | -0.000110531 | -2.77E-09 | 1.11E-04 |
| Hidden 1 Bias | -1.53E-13 | -1.53E-13 | 1.38E-16 |
| Hidden 2 Bias | -5.10E-05 | -5.11E-05 | 6.94E-08 |
| Hidden 2 to Output weight 1 | 0.001362537 | 0.001363 | 3.12E-08 |
| Hidden 2 to Output weight 2 | -5.39E-05 | -5.39E-05 | 3.19E-08 |
| Output Bias | -0.000831012 | -0.00095 | 1.19E-04 |

## 3.3 Training and Tanh activation function Method

In order to setup the data for training we first split the data into train, validation, and test sets then applied z-score normalization, the mean of training set is subtracted from all data and then they are divided by the standard deviation of the training set. This way we converted our training set to zero mean and unit standard deviation distribution. Since we will be using cross entropy loss, we also converted targets to their one-hot-encoded versions. Next we gathered the hyperparemeters from the configuration file and initialized our weights and biases to a random value between -.01 and .01. Once this pre-processing was completed we began training. In order to improve learning process we used mini-batch gradient descent, that is we split the training set into several minibatches and updated the weights after iterating through each minibatch. At each epoch we reshuffled the data and regenerated mini-batches. As we cycled through each minibatch we ran a forward pass through the network and calculated the train loss and accuracy for each mini-batch. Next thorough back propagation we calculated the delta values at output and hidden unit. At every epoch, we took the average loss and accuracy of these train losses and accuracies of different minibatches. The equations to solve for delta values are as follows, $\delta_j = (t_j - y_j)$ if $j$ is an output unit, $\delta_j = g'(a_j) \sum_k \delta_k w_{jk}$ if $j$ is a hidden unit.

We then applied weight changes using the following equation:

$$w_{ij} \leftarrow w_{ij} + \alpha \delta_j z_i \tag{2}$$

However in order to speed up training we applied momentum in our update rule by setting a value $\gamma$ to 0.9. With this, we were able to capture a running average of the previous gradients, and thus driving the weights in a direction that leads faster to the optimum values. The new update rule above can now be expressed as follows.

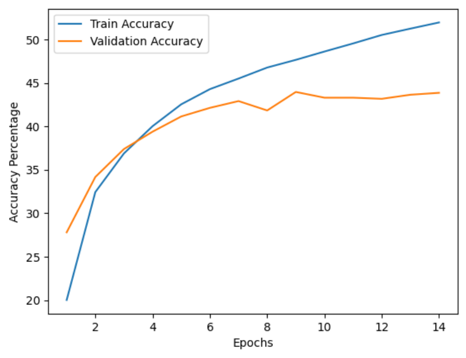
$$v_t = \gamma v_{t-1} + (1-\gamma) w_t \tag{3}$$

$$w_{t+1} = w_t - \alpha v_t \tag{4}$$

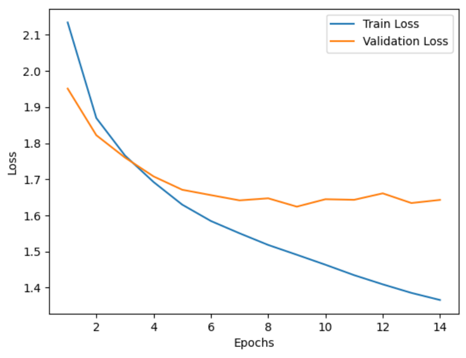
After iterating through all of the minibatches, we measured the validation loss and accuracy each epoch. Finally in order to prevent overfitting, we utilized early stopping. If validation set loss increases for 5 epochs then we stop the training and get the weights for the minimum validation loss. This saved a lot of time in training. In order to test our network for the first time we utilized the *tanh* activation function. This way, we kept the outputs of the layers(except for the output layer) between -1 and 1.

## 3.4 Training and Tanh activation function Results

The early stopping triggered at 13 epochs. We achieved at test accuracy of 44.02% and a test loss of 1.63. This was trained on a network of size 3072x64x64x10 with a learning rate of 0.005 and momentum of 0.9. (see figure 1)



(a) Accuracy using Tanh



(b) Loss with Tanh

Figure 1: Training on network of size 3072x64x64x10, $\gamma$= 0.9, LR = 0.005, epochs until stop = 13

### 3.5 Experiment with Regularization Methods

In order to reduce overfitting more, we implemented L2 regularization. This introduces a parameter $\lambda$ that when applied in the weight update equation, reduces over-fitting. With this method, we were able to make weights smaller in proportion to their size, i.e. penalizing bigger weights more than the smaller weights. The weight update equation is now equal to

$$w_{new} = w \cdot -\alpha \cdot \textbf{weight change} + (2 \cdot \lambda \cdot w) \tag{5}$$

$$b_{new} = b \cdot -\alpha \cdot \textbf{bias change} + (2 \cdot \lambda \cdot b) \tag{6}$$

### 3.6 Experiment with Regularization results

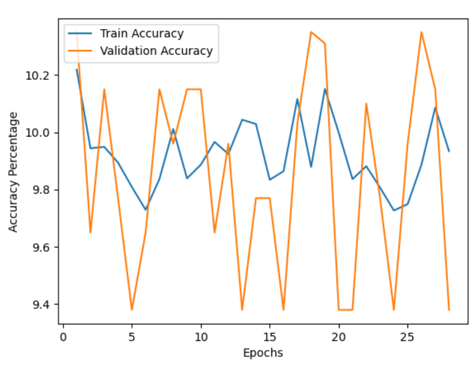
This section was trained on the same hyperparemeters as the training section but introduces the regularization value. However we did increase the early stopping value to 10 epochs. We found that with too low of a $\lambda$ value the network would not be able to train and we remain stuck at 10% accuracy. At $1e-3$ the model had no over-fitting but would not achieve as high of test accuracy. At $1e-4$ almost no change was seen with the regularization value. Therefore, it was found that $5e-4$ was the ideal regularization value.

Table 2: Test accuracy and loss for each regularization value

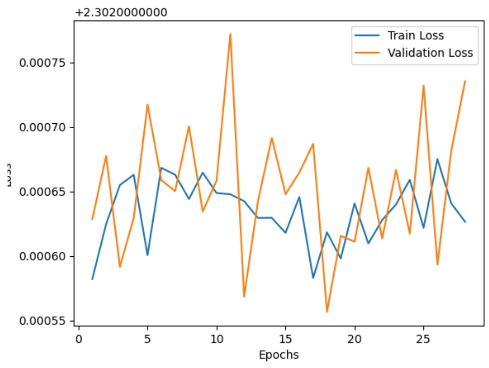
| Lambda | 1.00E-02 | 1.00E-03 | 1.00E-04 |
|---|---|---|---|
| Test Accuracy | 0.1 | 0.3994 | 0.4434 |
| Test Loss | 2.302 | 1.724 | 1.617 |

### 3.7 Experiment with Activations Methods

In the previous section we used Tanh as our activation function at each unit. In this section we experimented by testing with each of the following activation functions. Our hyperparameters for this section can be seen in table-3. Except for the Sigmoid function, we found that we had to use a learning rate of 0.05 in order to achieve any kind of performance. This is likely because the derivatives of the sigmoid are quite small compared to the other activation functions so it would require a larger learning rate to converge.

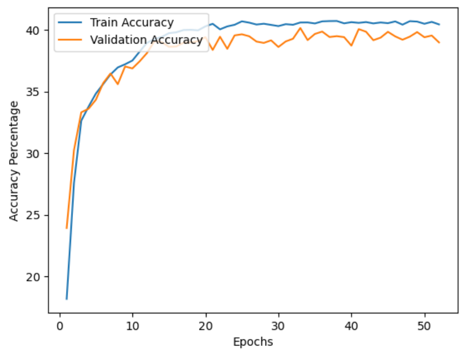


(a) Accuracy using Tanh and $\lambda$ = 1e-2

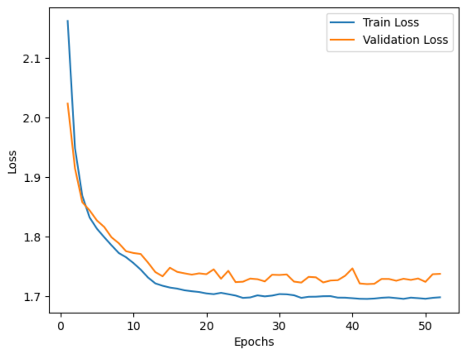


(b) Loss with Tanh and $\lambda$ = 1e-2

Figure 2: Network size 3072x64x64x10, $\gamma$= 0.9, LR = 0.005, epochs until stop = 28, $\lambda = 1e-2$

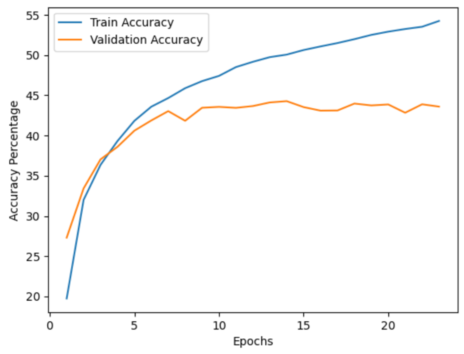


(a) Accuracy using Tanh and $\lambda$ = 1e-3

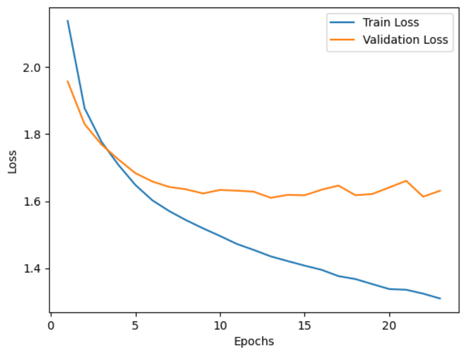


(b) Loss with Tanh and $\lambda$ = 1e-3

Figure 3: Network of size 3072x64x64x10, $\gamma$= 0.9, LR = 0.005, epochs until stop = 52, $\lambda = 1e-3$



(a) Accuracy using Tanh and $\lambda$ = 1e-4



(b) Loss with Tanh and $\lambda$ = 1e-4

Figure 4: Network size 3072x64x64x10, $\gamma$= 0.9, LR = 0.005, epochs until stop = 24, $\lambda = 1e-4$

Table 3: Hyperparameters for activation testing

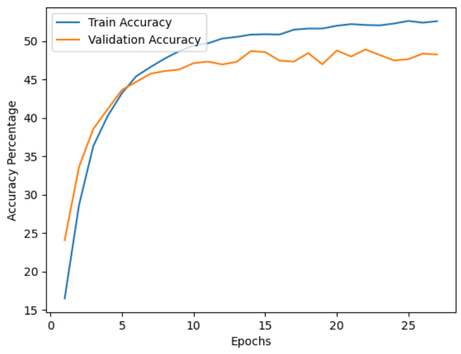
| | |
|---|---|
| Learning Rate | 0.005 (0.05 for sigmoid) |
| Stopping Criteria | 5 |
| Momentum | 0.9 |
| Lambda | 5.00E-04 |

## 3.8 Experiment with Activations Results

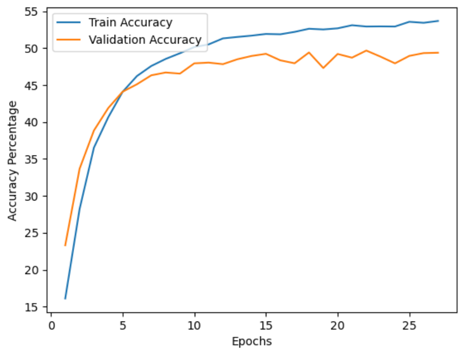
It is clear from the results that ReLU and Leaky ReLU are much better activation functions for this problem. The Sigmoid required a much higher learning rate and this is likely because it is susceptible to vanishing gradients. The test set and training set outcomes are as follows.

Table 4: Test set statistics of each of the activation functions

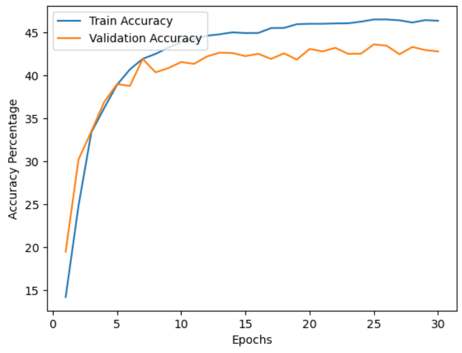
| Relu | | | LeakyRelu | | | Sigmoid | |
|---|---|---|---|---|---|---|---|
| Test Loss | 1.431 | | Test Loss | 1.441 | | Test Loss | 1.621 |
| Test Accuracy | 50.41% | | Test Accuracy | 49.35% | | Test Accuracy | 42.68% |



(a) Accuracy using Leaky ReLU

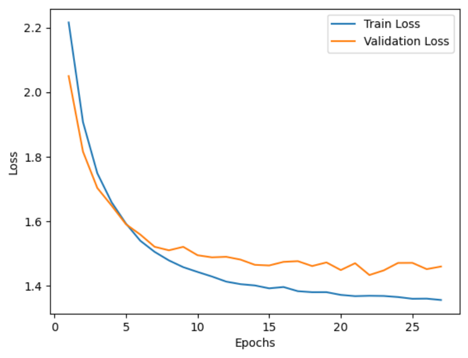


(b) Accuracy using ReLU

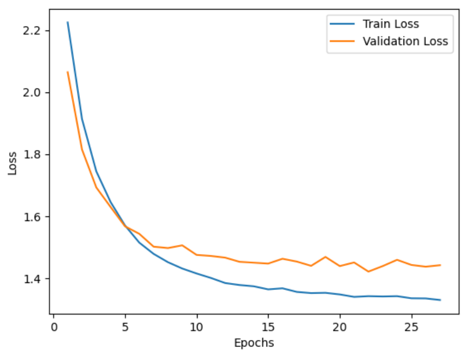


(c) Accuracy using sigmoid

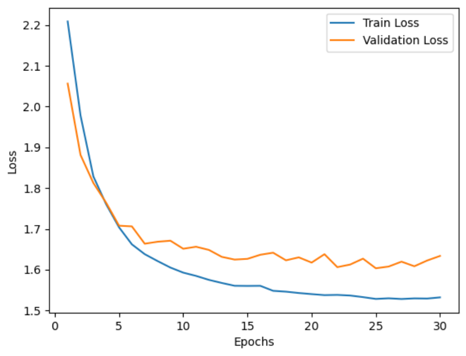
Figure 5: Network size 3072x64x64x10, $\gamma$= 0.9, LR = 0.005, 0.05 for sig, $\lambda = 5e - 4$



(a) Loss using Leaky ReLU



(b) Loss using ReLU



(c) Loss using sigmoid

Figure 6: Network size 3072x64x64x10, $\gamma$= 0.9, LR = 0.005, 0.05 for sig, $\lambda = 5e - 4$

### 3.9 Experiment with Network Topology Methods

In this section we experimented by changing the architecture of the Neural network. We found from the previous section that the ideal hyperparameters are as follows:

Table 5: Hyperparameters used for topology experiments

| | Parameter |
|---|---|
| Activation function | ReLU |
| Learning Rate | 0.005 |
| Stopping Criteria | 5 |
| Momentum | 0.9 |
| Lambda | 5.00E-04 |

#### 3.9.1 Halving and Doubling the Hidden Units Methods

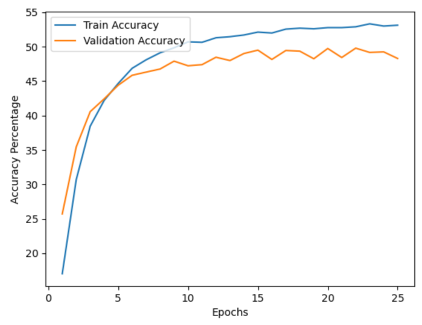
We tried changing the amount of hidden units per layers. We tried the following combinations; $64 \times 128$, $128 \times 64$, $32 \times 64$, and $64 \times 32$. The hyper-parameters used in this experiment can be seen in table-5.

#### 3.9.2 Halving and Doubling the Hidden Units Results

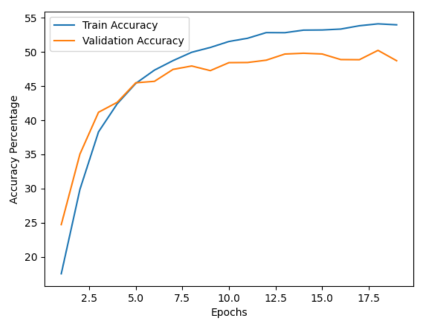
Out of each of these the performance seemed to go down when we reduced the amount of hidden units or made the second hidden layer larger. When we made the first hidden layer larger it had about the same performance. It is likely that 64x64 hidden units is more than enough units to extract the information from the images so adding or subtracting more did not effect it.

Table 6: Test Loss and Accuracy of varying hidden units in each layer

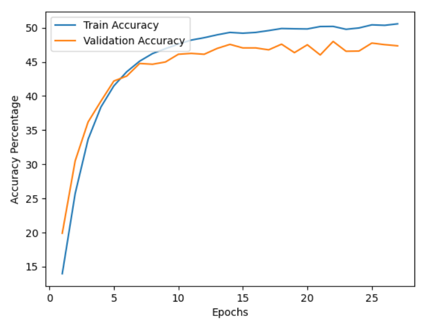
| | 64x128 | 128x64 | 32x64 | 64x32 |
|---|---|---|---|---|
| Test Loss | 1.453 | 1.41 | 1.472 | 1.449 |
| Test Accuracy | 48.53 | 50.24 | 48.17 | 48.48 |



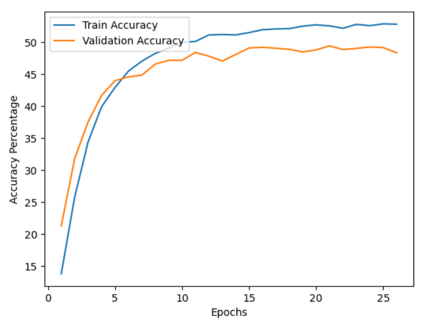
(a) Accuracy of 64x128 hidden units



(b) Accuracy of 128x64 hidden units



(c) Accuracy of 32x64 hidden units



(d) Accuracy of 64x32 hidden units

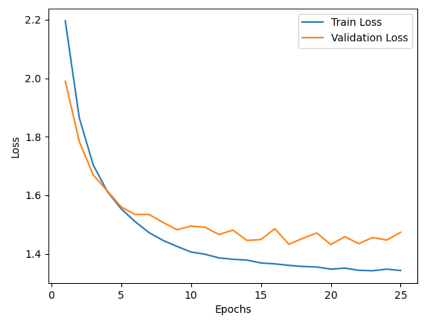
Figure 7: Network with varying hidden units, $\gamma$= 0.9, LR = 0.005, $\lambda = 5e - 4$

#### 3.9.3 Increasing the number of hidden layers methods

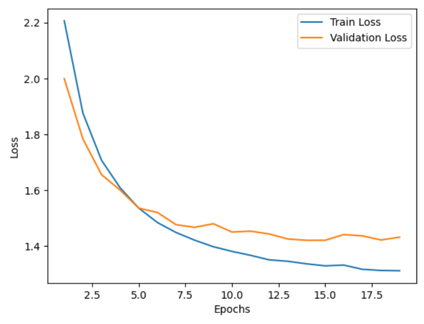
This section discusses increasing the total number of hidden layers. Expectation was that increasing the number of hidden layers will make the network learn better, resulting in a lower training loss. Interestingly it was found that by only increasing the number of layers, the network would not train properly. In order to fix this we had to increase the learning rate from 0.005 to 0.04. This is the only hyper-parameter that is different from table 5. The total number of hidden units were selected as it results in about the same amount of weights and biases as the original 64x64 network.

#### 3.9.4 Increasing the number of hidden layers results

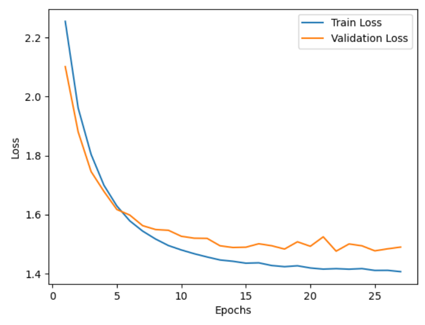
It was found that increasing the number of hidden layers only reduced the overall test accuracy of the network. This may be because of overfitting that came with the increased number of hidden layers or it is likely because there is not enough information in the images to extract that would require so many layers. Additionally, by increasing the total number of units in the first hidden layer and reducing the number of units in the following 2 hidden layers greatly reduced the accuracy. It is important to note that in order to properly train these networks the learning rate had to be increased. This could be because the optimization surface is much more complex with more layers and it is more prone to get stuck in a local minima.



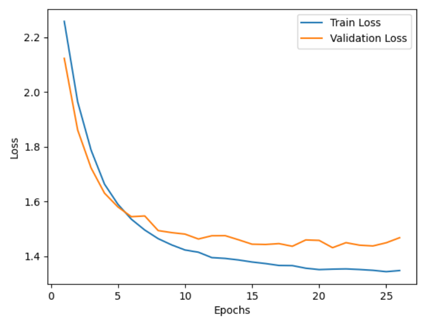
(a) Loss of 64x128 hidden units



(b) Loss of 128x64 hidden units



(c) Loss of 32x64 hidden units



(d) Loss of 64x32 hidden units

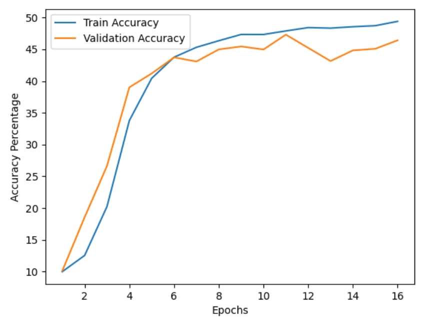
Figure 8: Network with varying hidden units, $\gamma$= 0.9, LR = 0.005, $\lambda = 5e - 4$

Table 7: Test Loss and Accuracy from increasing the number of hidden layers

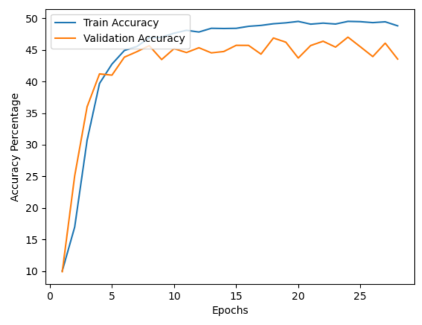
| Network size | 64x32x32 | 72x24x24 |
|---|---|---|
| Test Loss | 1.5008 | 1.583 |
| Test Accuracy | 47.09 | 44.18 |

### 3.9.5 Decreasing the number of hidden layers methods

This section involves reducing the total number of hidden layers to one. The hyperparameters are the ones used in table 5. The single hidden layer had 69 hidden units as this is the closest amount for the total weights and biases of the original network.

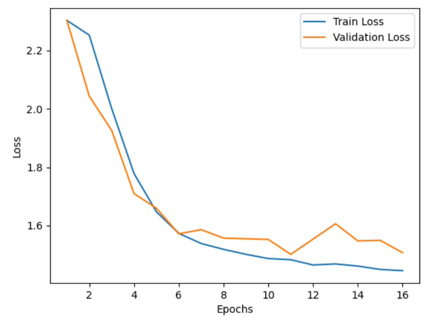


(a) Accuracy of 64x32x32 hidden units

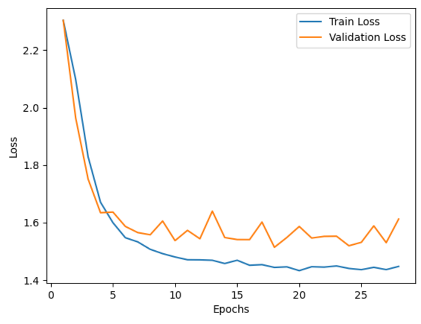


(b) Accuracy of 72x24x24 hidden units

Figure 9: Networks with varying hidden units, $\gamma$= 0.9, LR = 0.04, $\lambda = 5e - 4$



(a) Loss of 64x32x32 hidden units



(b) Loss of 72x24x24 hidden units

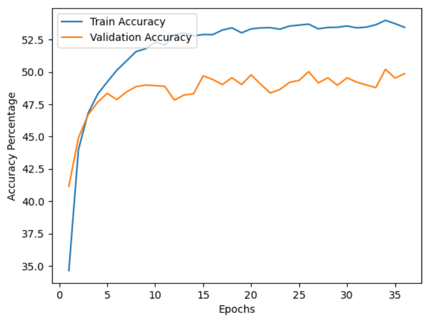
Figure 10: Network with varying hidden units, $\gamma$= 0.9, LR = 0.04, $\lambda = 5e-4$

#### 3.9.6 Decreasing the number of hidden layers results

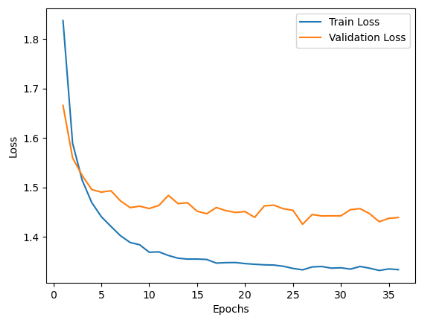
It was found that changing the total width and depth of the network does not improve the overall performance. The test and train results is as follows.

Table 8: Test Loss and Accuracy from decreasing the number of hidden layers

| Network size | 69 units |
|---|---|
| Test Loss | 1.4338 |
| Test Accuracy | 49.4 |



(a) Accuracy of 69 hidden units



(b) Loss of 69 hidden units

Figure 11: Network with varying hidden units, $\gamma$= 0.9, LR = 0.005, $\lambda = 5e-4$

## 4 Discussion of results

After running these experiments it was found that the one change that had the greatest effect on improving overall performance was selecting the correct activation function. Regularization helped reduce over-fitting but did not increase accuracy. Changing the width and depth of the network had no change or a negative impact on the overall performance.

### 4.1 Individual contributions to project

#### 4.1.1 Yigit

My main responsibility in this project was to create overall structure of the network as well as readme file. I implemented the whole forward and backward pass process with early stopping and minibatch system in the network, and gradient descent with momentum to update the weights. I also implemented a function to automatically check part-b of the assignment. Finally, I wrote the final draft of the report.

#### 4.1.2 Marcus

I was responsible for making sure the code passed all of the sanity checks and making sure that the gradient was computed correctly. I also wrote code for preprocessing and assigning samples to sets. I also setup and tested all of the activation functions. I ran all of the test for the different network changes and collected the data for each step of the project. In order to do this I wrote the plotter code. Finally, I wrote the first draft of the report.

References

[1] A. Krizhevsky, V. Nair, and G. Hinton, CIFAR-10 and CIFAR-100 datasets, 2009. [Online]. Available: https://www.cs.toronto.edu/ kriz/cifar.html. [Accessed: 05-Feb-2022].

[2] P. Savarese and M. Maire, “Learning implicitly recurrent cnns through parameter sharing,” arXiv.org, 13-Mar-2019. [Online]. Available: https://arxiv.org/abs/1902.09701. [Accessed: 05-Feb-2022].