# CSI 4140/5140 — Project 1 Report

Group Number 7
Lucas Costello - G00830797

## 1 Problem Statement

Creating a robust deep neural network (DNN) that can achieve a classification accuracy of 80% or above for the CIFAR-10 dataset is not a simple feat. This task requires us to build a neural network entirely from scratch, with an additional requirement that we don't use any of the specified PyTorch functions. By not using any of the pre-made functions we get to showcase our understanding of the low-level implementation of neural networks. Achieving a high classification accuracy isn't the only objective for this project. After creating the model, we will conduct an ablation study that evaluates the effects of different optimization strategies, regularization techniques, and learning rate decay methods.

This task is compelling because CIFAR-10 remains a classic dataset for image recognition. It is used world wide by researchers and students to experiment and validate key machine learning concepts. Building each layer and function from scratch allows us to understand how our design choices impact the results of a model. This goes hand in hand with hyper-parameter testing, allowing us to figure out the most optimal settings for our specific model design. The knowledge and experience gained from this will be crucial towards moving beyond black-box models and enabling us to create and tune neural networks for other real world applications.

## 2 Design and Implementation

### 2.1 Model Architecture

To achieve the task of reaching a classification accuracy of above 80% we chose to create a neural network based on the ResNet-34 architecture. Our model consists of numerous layers, but the initial layer is the first. In this layer, the input goes through a convolution, batch normalization, and then a ReLU activation function. The purpose of the initial layer is to extract basic patterns of the image (i.e edges, textures, etc.) and increase the number of channels so that the network can learn more features. After the initial layer, there are four more subsequent layers containing residual blocks that extract even more features.

The make-up of the residual block is the same across layers one through four, with an exception that layer 1 has a skip connection and the others all have a downsample connection. Our residual block is composed of two cycles of convolution, batch normalization, and a ReLU activation function. This is followed by a third convolution and batch normalization, along with a squeeze and excitation block followed by a dropout. The first two cycles extract features from the input. Then the next convolution and batch normalization layer prepare the input for the squeeze and excitation block, where important features are emphasized and less useful ones are suppressed. After that, the model is regularized again through dropout. Since there is not a risk of mismatched sizes, layer 1 utilizes a skip connection. This means that after going through the residual block, the initial 32 channel input is added to the output without a convolution. That is then sent through another ReLU activation function, which leads to the consecutive layer.

After the initial layer, the input goes through layers 1, 2, 3, and 4. These layers output 32, 64, 128, and 256 channels, and consist of 3, 4, 6, and 3 residual blocks respectively. As stated prior, layer 1 utilizes a skip connection. Layers two through four utilize a downsample connection that matches the input size to the output size, ensuring that their dimensions are a match. The downsample comes into play prior to the residual addition. After the fourth layer, the output is pooled through adaptive average pooling. This reduces the dimensions of the feature maps to a fixed size. The output from the pooling is flattened and then regularized through dropout. After that, it goes to the fully connected layer, mapping the feature representations to the final output classes. Those logits from the fully connected layer go through a SoftMax function, converting them into probability distributions over the classes.

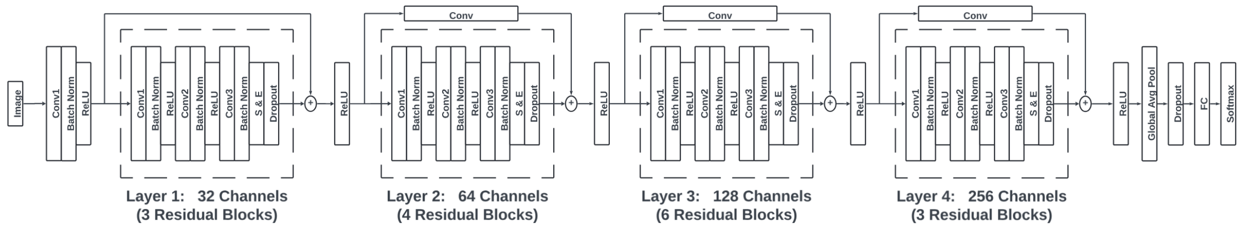

Figure 1: Model Architecture Diagram

## 2.2 Methods

The features and methods we employed are specifically designed to enhance the performance of our model for the CIFAR-10 image classification task. The first feature we implemented was data augmentation. During training, our data goes through several transformations: random cropping, random horizontal flipping, random rotations, and color jitter. These transformations allow the model to distinguish features regardless of direction, orientation, color, and position. After that, the images are normalized using the mean and standard deviation of the CIFAR-10 dataset. All of this is useful for stabilizing and accelerating the training process. The test data is separate from the training data, only being normalized, without any augmentation, in order to promote consistency in our evaluation of the model over each epoch.

The architecture of our model is based on the ResNet framework. It consists of multiple layers of residual blocks that utilize skip connections. The use of these skip connections is that they facilitate gradient flow, mitigating the vanishing gradient problem for us. Inside of each residual block we implemented a Squeeze-and-Excitation (SE) block. The purpose of these blocks is to emphasize the features that the model determines important, while subduing the ones that aren't. For the task of classifying images this SE block will improve our accuracy by making it better at recognizing key patterns and details. Beyond incorporating the SE block, we've added a few more features to enhance our model: dropout regularization, adaptive average pooling, and a SoftMax activation function. Dropout helps prevent overfitting by randomly deactivating neurons during training, which makes the model more robust. We utilize adaptive average pooling to resize the feature maps to a consistent size before sending them to our fully connected layer (FC). The FC layer maps everything to the 10 classes that make up CIFAR-10. While training the model we use the raw logits, but during evaluation we employ a SoftMax function to map those logits into probabilities. This makes the predictions of the model easier to interpret.

To further prevent overfitting, our model also uses L2 regularization (weight decay). The purpose of this is to penalize large weights within the model, ensuring stability and generalization. To achieve a high classification accuracy, the optimization strategy that we selected was Stochastic Gradient Descent (SGD) with momentum. The use of momentum in our optimizer helps accelerate the convergance of the model by stabalizing and smoothing the updates. After our learning rate is initialized we use a step scheduler to reduce it after a set number of epochs. Using a step scheduler allows for larger updates in the initial phases of training and then fine tuning in the later phases, after lowering the learning rate.

## 3 Evaluation

The result of a greater than 90% classification accuracy of the CIFAR-10 dataset was only achieved after rigorous testing with variations of the hyper-parameters. We ran trials using the Adam, Stochastic Gradient Descent, and RMSprop optimization algorithms. For each of the algorithms, the following learning rate decay algorithms were tested: step, exponential, and cosine annealing learning rate decay. Depending on the initial results for each test, another was ran with different regularization coefficients in order to gauge their effectiveness. The tests at the beginning were all ran for 200 epochs with a batch size of 128, but in later tests we increased the batch size to 256 and lowered the epochs to 100. Both ways allowed us to achieve a classification accuracy of 95% $\pm$ 2% but the ladder was more efficient.

In order to better understand how our model was performing, at the end of our code we create graphs. Each test output every one of these graphs: accuracy over epochs, loss over epochs, learning rate over epochs, and cost over iterations. By inspecting these we were allowed to visually see how our model was performing and figure out where to make improvements. The results of our rigorous testing were the optimal hyper-parameters for our model. Utilizing SGD with a momentum of 0.9 along with a cosine annealing learning rate decay algorithm allowed us to achieve our best classification accuracy of 94.93%. Additionally, L2 regularization was utilized in the optimizer, but we found that a weight decay coefficient of 0.0005 worked better than lambda for our model. For the decay algorithm we set $T_{max}$ equal to the number of epochs, allowing the learning rate to gradually decay over the duration of the whole run. To achieve that accuracy we did run it for 200 epochs, but by lowering it to 100 epochs we still achieved an accuracy of 93.51

## 4 Ablation Study

After achieving the required accuracy, we ran more tests specifically changing certain hyper-parameters and algorithms to determine their effect. Below in figures 2,3, and 4 visual representations can be seen of the learning rates for each of the different decay algorithms.

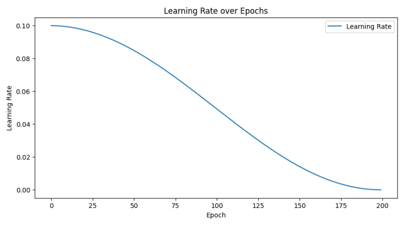

Figure 2:
Cosine Annealing LR Decay

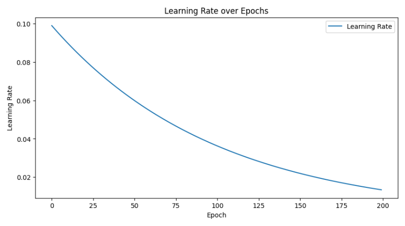

Figure 3:
Exponential LR Decay

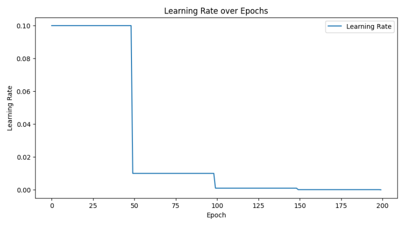

Figure 4:
Step LR Decay

In order to test the effect of the learning rate decay algorithms we kept SGD with momentum and all other hyper-parameters as a constant. As can be seen below, each algorithm has its merits and demerits. The cosine annealing and exponential learning rate decay algorithms have very similar effects on the training and test accuracy of our model. It is worthy to note that although they both have a relatively smooth convergence, there is a visible deformity in the training accuracy while utilizing the cosine annealing algorithm. Due to the nature of the two algorithms, how they both gradually decay the learning rate, it is not surprising that they have similar effects on the models performance. Notably different from those two is the step learning rate decay algorithm. Where the others gradually decay, this one does not. Every 50 epochs the learning rate was decreased and it had an interesting effect. Within the first 50 epochs the test accuracy was sporadic, but after that both the training and test accuracy smoothed out. It can be noted that utilizing this algorithm caused the accuracy of the model to plateau.

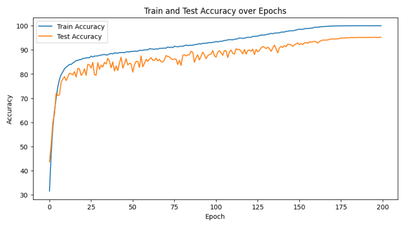

Figure 5:
Training & Test Accuracy
Cosine Annealing Decay

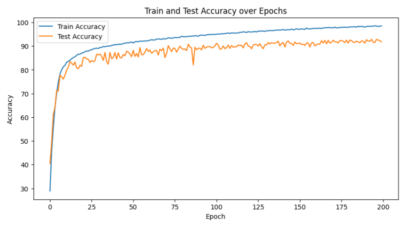

Figure 6:
Training & Test Accuracy
Exponential Decay

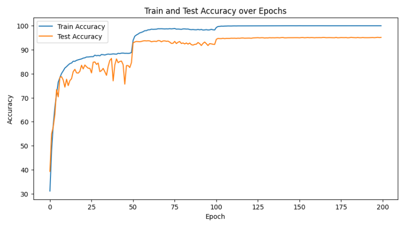

Figure 7:
Training & Test Accuracy
Step Decay

The trial on the initial learning rate was also completed with SGD with momentum. For this we tested learning rates of 0.1 and 0.2 to see how the model was affected. Below are *Figures 8 & 9* that give a visual representation of the training and test accuracy for each hyper-parameter, while *Figures 10 & 11* showcase the loss. First, each test exceeded 90% test accuracy with the smaller initial learning rate achieving 95.14% and the larger achieving 94.22%. In both cases the accuracy and loss seem to plateau after about 100 epochs. Notably, the lower initial learning rate seems to have a much smoother time converging than the ladder. This can be seen by how sporadic the test accuracy is in the given figures. What we can gather from this is that, using SGD with momentum, 0.2 is too high of an initial learning rate to provide good results for my model. We only tried the two different initial learning rates, but if given more time would like to try 0.15, 0.05 and 0.01.

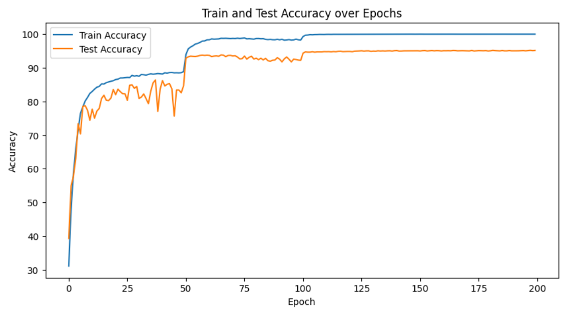

Figure 8:
Training & Test Accuracy
(LR = 0.1)

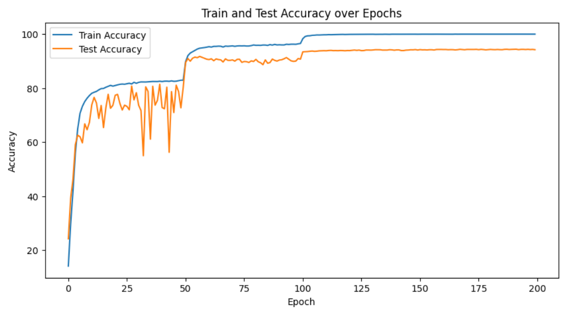

Figure 9:
Training & Test Accuracy
(LR = 0.2)

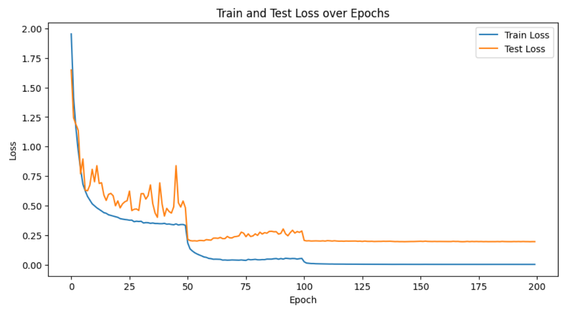

Figure 10:
Test & Training Loss
(LR = 0.1)

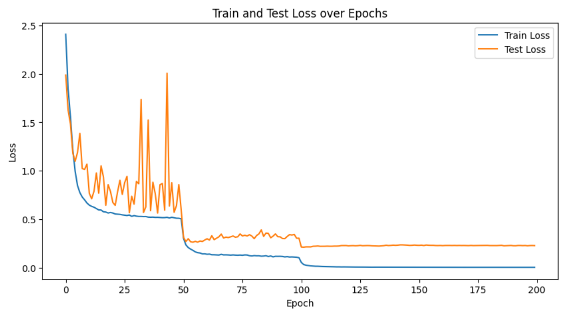

Figure 11:
Test & Training Loss
(LR = 0.2)

Another parameter that was tested was the number of training epochs. For most of the tests, 200 was selected as the number of epochs. To compare, we continued to use SGD with momentum to test our model for 100 and 250 epochs. All three of these tests were done with the cosine annealing learning rate decay algorithm. From a glance, it seems as though running over 100 epochs was kind of pointless with our model. After 100, 200, and 250 epochs the respective test accuracies were 94.24%, 95.17%, and 95.73%. With nothing else changing except the number of epochs, we can say that the effect of the number of epochs is minimal after 100. With that being said, changing other parameters such as the learning rate alongside the number of epochs would probably lead to a more substantial difference in outputs.

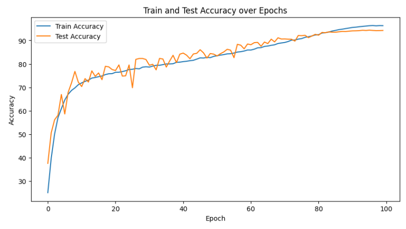

Figure 12:
Training & Test Accuracy
100 Epochs

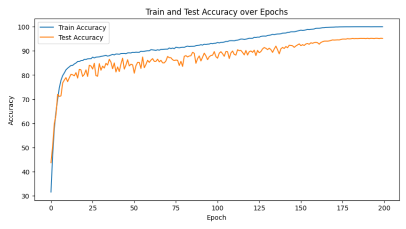

Figure 13:
Training & Test Accuracy
200 Epochs

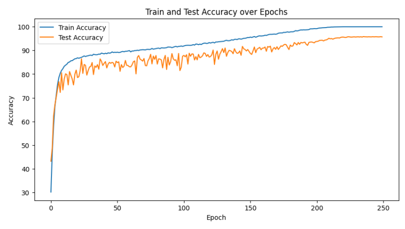

Figure 14:
Training & Test Accuracy
250 Epochs

To determine the effect of Lambda L2 regularization, tests were conducted using both Adam and SGD with momentum as the optimizer. We did originally test 0.0001 as the value for Lambda L2, but thorough testing revealed that 0.0005 was a better weight decay value. As can be seen below, there is quite a difference visually regarding the implementation of lambda. For SGD with momentum, the convergence seems to be smoother without L2 rather than with it. That being said, when it does not have L2 it seems to plateau faster. When utilizing L2, although the test accuracy is more sporadic it continues to increase over the course of 200 epochs, only appearing to plateau towards the very end. For Adam, the results seem to be similar. When utilizing L2 regularization with Adam, we can see that it actually causes some disparity between the training and test accuracy. Without L2, the Adam optimizer seems to be more accurate.

Another regularization technique utilized was dropout. Again, SGD with momentum and Adam were used to evaluate this hyper-parameters effect. It is notable in the graphs that without dropout there is a decent sized gap between the training accuracy and the test accuracy. The first test done on both optimizers was with a dropout probability of 0.5. With this probability the training and test accuracy were much more consistent over the epochs. The other dropout probability that we tested was 0.3. The point of this test was to lower the amount of regularization to see its effect. On SGD with momentum, dropping 30% of the neurons created a noticeable difference in the test accuracy over each epoch, becoming more sporadic while still following the training accuracy curve. With Adam, changing the probability between 0.5 and 0.3 did not really affect it much with those probabilities resulting in a test accuracy of 94.5% and 94.02% respectfully.

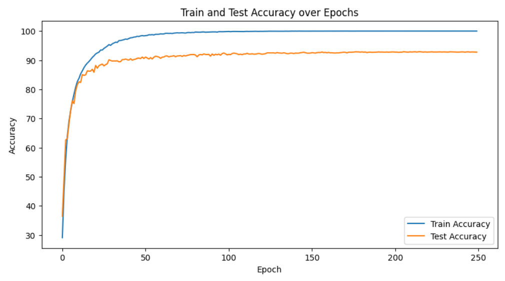

Figure 15:
Training & Test Accuracy
SGD w Momentum
No L2 Regularization

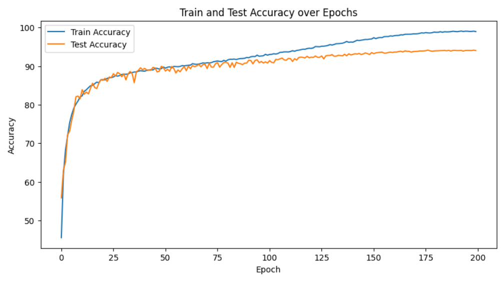

Figure 16:
Training & Test Accuracy
Adam
No L2 Regularization

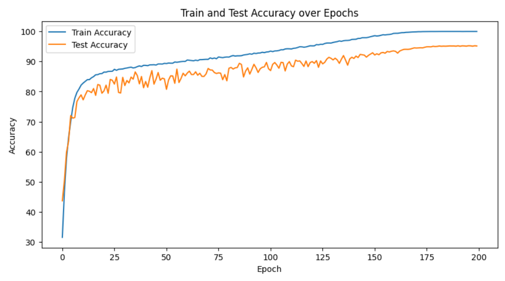

Figure 16:
Test & Training Loss
SGD w Momentum
L2 Regularization

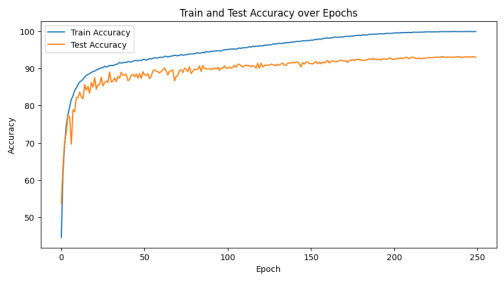

Figure 17:
Test & Training Loss
Adam
L2 Regularization

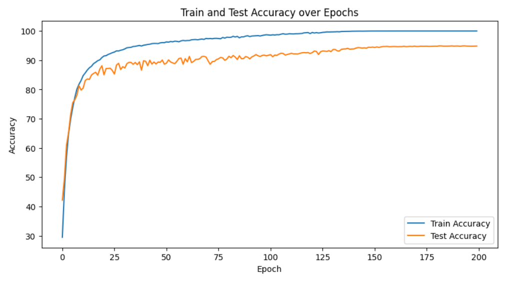

Figure 17:
Training & Test Accuracy
SGD w Momentum
No Dropout

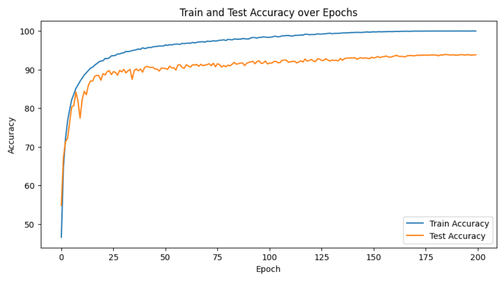

Figure 18:
Training & Test Accuracy
Adam
No Dropout

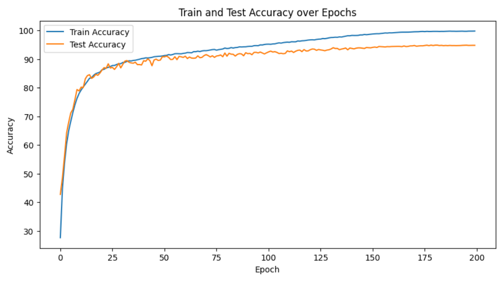

Figure 18:
Test & Training Loss
SGD w Momentum
Dropout (0.5)

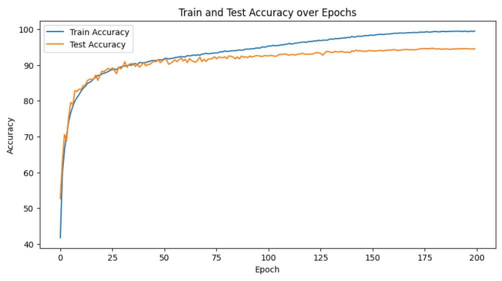

Figure 19:
Test & Training Loss
Adam
Dropout (0.5)

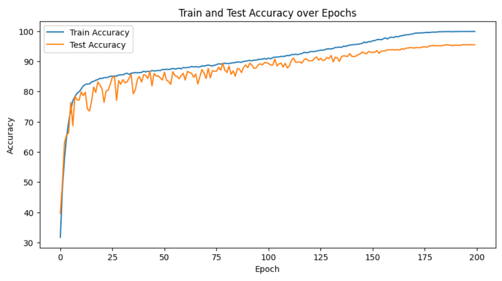

Figure 19:
Test & Training Loss
SGD w Momentum
Dropout (0.3)

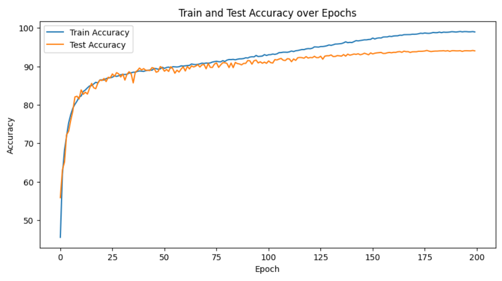

Figure 20:
Test & Training Loss
Adam
Dropout (0.3)

In order to maximize the robustness and generalization ability of our model we implemented numerous data augmentation techniques to the training image set. Specifically, we applied random cropping, horizontal flips. rotation, affine, color jitter, and normalization. Random cropping with padding was used to increase the diversity of spatial locations from which features can be learned. This in turn makes our model more resilient to shifts in the input images. Random horizontal flipping provides invariance to the horizontal orientation of the images, which makes the model better equipped to recognize objects regardless of their left and right positioning. Random rotation allows the model to handle rotational variations of up to 15 degrees. This further improves our models ability to generalize across rotated instances. The random affine transformation incorporates random translations, scaling, and shear distortions, which force our model to learn features that are more robust to spatial deformations. Color jitter was used to change the brightness, contrast, saturation, and hue of the training images. Using color jitter allows our model to learn to focus on shapes and structures rather than color variations. After all of that we converted the images to tensors and normalized them so that the training process remained stable despite all of our transformations.

Through the numerous tests of differing hyper-parameters it was probably noted that RMSprop has not yet made an appearance. There is a reason for that. Below are graphs showcasing the effect of RMSprop as an optimizer on our model. As can be seen in the graphs, this optimizer produced the lowest training and test accuracy out of any of the optimization methods. First, RMSprop was tested with and without dropout. The results of those tests were training accuracies of under 90% and test accuracies in the mid to high 80s. We also tested RMSprop with a dropout probability of 0.3 and it resulted with a training accuracy of 72.26% and a test accuracy of 74.87%. These were so underwhelming compared to the initial results of other optimization methods that we did no further tests on this optimizer.

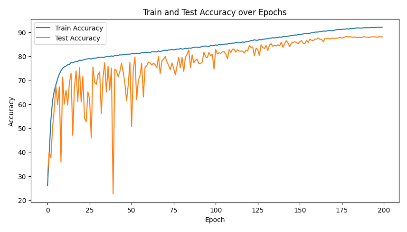

Figure 21:
Training & Test Accuracy
RMSprop
No Dropout

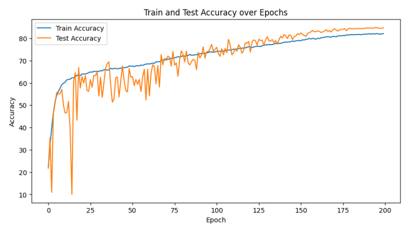

Figure 22:
Training & Test Accuracy
RMSprop
Dropout (0.5)

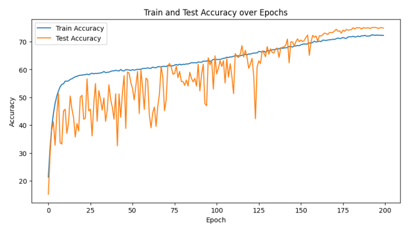

Figure 23:
Training & Test Accuracy
RMSprop
Dropout (0.3)

The next hyper-parameters that were tested were the beta values used by the Adam optimizer. Initially, we ran tests using Adam with beta values of 0.9 and 0.999 for beta1 and beta2 respectfully. This was our baseline for the next couple tests. As can be seen with the base values Adam has a good, steady convergence. After changing the beta2 value to 0.97, it can be observed that the test accuracy actually decreases over the duration of the epochs. After about 5-15 epochs there is a sharp decline in test accuracy that never seems to catch back up. Changing both beta values did not affect the model like we thought it would.

Decreasing beta1 to 0.7 and beta2 to 0.9 resulted in a similar test accuracy to just changing beta2. Regarding this hyper-parameter, we do not see a reason why it would be changed from its initial values. They seem to work the best, with all others being less accurate and less optimal for our model.

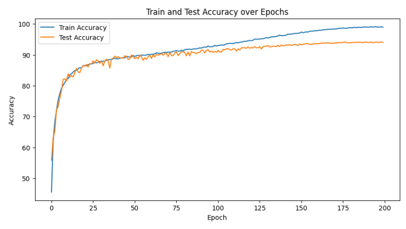

Figure 24:
Training & Test Accuracy
Adam
(0.9, 0.999)

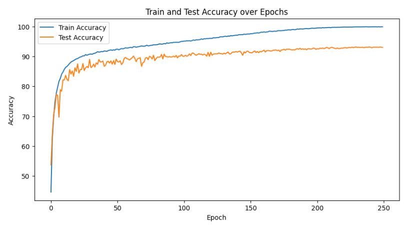

Figure 25:
Training & Test Accuracy
Adam
(0.9, 0.97)

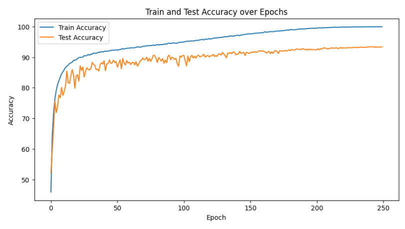

Figure 26:
Training & Test Accuracy
Adam
(0.7, 0.9)

## 5 Individual Contributions

Considering that I went solo for this project there are no contributions to give besides my own. I did every test on my own and wrote the report by myself. Overall, this was a very fun and engaging project. It definitely smoothed out my edges in terms of understanding neural networks. Throughout this project I had to research and figure most of the problems out on my own. While taking more time and effort on my end, I do believe that it was worth it.

## 6 Sources

- He, Kaiming, et al. Deep Residual Learning for Image Recognition. 10 Dec. 2015, arxiv.org/pdf/1512.03385.
- He, Tong, et al. Bag of Tricks for Image Classification with Convolutional Neural Networks. 5 Dec. 2018, arxiv.org/pdf/1812.01187.
- Hu, Jie, et al. Squeeze-And-Excitation Networks. 16 May 2019, arxiv.org/pdf/1709.01507.
- “Pytorch.” GitHub, github.com/pytorch.